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 geralmente apresentam limitações no número de linguagens suportadas, nas opções de configuração e não permitem a definição de estilos de formatação personalizados. Visando superar ou reduzir as limitações frequentemente encontradas nas ferramentas convencionais de formatação, neste trabalho é apresentado o *Witch Cooking*, um software prototípico que tem como objetivos: (I) abranger uma gama de linguagens de programação, além de (II) permitir que o usuário defina seus próprios estilos de formatação.

Palavras-chave: formatação de código; *prettyprint*; formatação personalizada; *Tree-Sitter*; linguagem de consulta.

Lista de Ilustrações

Lista de Trechos de Código

Sumário

1 Introdução

A flexibilidade presente na sintaxe de linguagens de programação permite que diferentes arranjos de um 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. 546; 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 opinativas, assim restringindo as possibilidades de personalização pelo usuário. Finalmente, é também comum que essas ferramentas de formatação não permitam que o usuário defina estilos de formatação personalizados.

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, que são frequentemente presentes nas ferramentas convencionais de formatação, este trabalho tem como objetivo geral desenvolver um software prototípico para a formatação de código. A ferramenta desenvolvida, nomeada de *Witch Cooking* (SILVA, 2023b), tem como objetivos: (I) abranger uma gama de linguagens de programação, além de (II) permitir que o usuário defina seus estilos personalizados 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 a árvore 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]: (I) uma responsável por produzir uma sequência de caracteres que representavam o programa sendo impresso; e (II) outra 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.

Recentemente, tem se tornado cada vez mais evidente a crescente relevância dos softwares opinativos (ECCLES, 2015), também conhecidos em inglês como *opinionated softwares*. Esses softwares preestabelecem práticas estritivas, visando deliberadamente frustrar ou dificultar práticas destoantes (LANCE, 2021). Similarmente, os *prettyprinters* primitivos, tais como o SOAP (SCOWEN et al., 1971), eram naturalmente restritivos devido à simplicidade e às limitações dos algoritmos de formatação existentes. Conforme mencionado, à medida que os algoritmos de formatação de código-fonte eram aprimorados, tornando-se mais robustos e sofisticados, os *prettyprinters* passaram a proporcionar mais personalizações aos usuários. No entanto, atualmente, a abordagem opinativa também tem sido adotada por softwares modernos de formatação de código-fonte (BLACK…, 2023; GRIESEMER, 2022, p. 5–6, 8). Esses softwares geralmente têm como objetivos: (I) padronizar e (II) garantir a consistência do código; além (III) eliminar o tempo e esforços necessários para escolher os estilos ideais de formatação; e, finalmente, consequentemente e principalmente (IV) aprimorar o ambiente de desenvolvimento colaborativo.

Um exemplo proeminente de *prettyprinter* opinativo é o ***Black*** (BLACK…, 2023), um formatador de código Python, cuja logomarca parodia a da fabricante automobilística Ford, e cujo irônico slogan ***"any color you like"*** faz alusão à famosa frase *"any customer can have a car painted* ***any colour that he wants so long as it is black****"* (FORD; CROWTHER, 1922, p. 72, grifo nosso), dita por Henry Ford, o fundador da fabricante. Assim como Ford adotou uma abordagem de produção em massa na fabricação dos *Model Ts* com uma única opção de cor para aumentar a eficiência (FORD; CROWTHER, 1922), o *Black* segue um princípio análogo. Ao impor um estilo estrito de formatação, minimizando o espaço para personalização ou preferências individuais, o *Black* oferece velocidade, determinismo, além de economia de tempo e esforços para questões mais importantes (BLACK…, 2023).

Sumarizando, desenvolver um *prettyprinter* não é uma tarefa simples (HUGHES, 1995, p. 55). Durante o processo de formatação, é necessário (I) ter conhecimento das estruturas gramaticais da linguagem específica em questão. Além disso, o objetivo do *prettyprinter* é (II)

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

formatar o código de maneira otimizada e agradável para o leitor (HUGHES, 1995, p. 55), 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 que atenda 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 (JASPER, 2023, p. 12; GOLDSTEIN, 1973; HUGHES, 1995; YELLAND, 2015a). 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.

3 O Sistema *Tree-Sitter*

O *Tree-Sitter* é um sistema multilíngue 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, consequentemente, são suportados pelo sistema.

Na análise de código, é comum realizar tarefas que envolvem a busca de padrões na árvore sintática. Para isso, o *Tree-Sitter* oferece uma pequena linguagem de consulta declarativa que é capaz de expressar esses padrões por meio de *S-expressions* 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 permitido o uso de predicados — funções arbitrárias geralmente utilizadas para filtrar nós ou realizar verificações mais complexas durante a busca de padrões —, sejam eles *builtins* 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 de código.

Em resumo, o sistema *Tree-Sitter* se mostra 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 da árvore sintática 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 mostra 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*

4.1 Motivações e Propósitos

As motivações para o desenvolvimento do *Witch Cooking* surgiram das dificuldades enfrentadas pelo autor ao estudar estilos de formatação de código-fonte em diversas linguagens de programação com as quais ele trabalhava. O estudo envolvia a exploração e análise tanto de estilos convencionais[1], dos quais muitos eram amplamente adotados, quanto de estilos não convencionais. Durante a análise dos estilos convencionais de formatação, (I) era imprescindível recorrer a diferentes *prettyprinters*, (II) cada um com suas próprias configurações e níveis de suporte para esses estilos. Por outro lado, durante a análise dos estilos não convencionais, (III) a aplicação manual era inevitável, consumindo consideravelmente tempo e esforço.

Tendo em vista (I) as dificuldades mencionadas anteriormente, além da (II) complexidade envolvida no *prettyprinting* — (III) agravada pela proposta de formatação multilíngue —, o *Witch Cooking* foi concebido como um software de linha de comando — de natureza prototípica — para a formatação de código-fonte. A ferramenta elaborada tem como objetivos: (I) abranger uma gama de linguagens de programação e (II) permitir que o usuário defina seus próprios estilos de formatação de código-fonte.

4.2 Usagem

O *Witch Cooking* apresenta a seguinte usagem `cook [-l LANG] -q QUERY [SRC]`, onde: (I) `cook` é o nome do executável, (II) `[-l LANG]` é uma opção para explicitar a linguagem do código-fonte a ser formatado, (III) `-q QUERY` é utilizado para definir o caminho para o arquivo — escrito na linguagem de consulta do *Tree-Sitter* — no qual as instruções de formatação estão definidas, e (IV) `[SRC]` é um argumento opcional para especificar o caminho para o arquivo a ser formatado. Quando `[SRC]` é provido e `[-l LANG]` é omitido, o software tentará inferir uma linguagem conforme a extensão do nome do arquivo. Por outro lado, quando `[SRC]` é omitido, o código-fonte a ser formatado deve ser provido via *pipeline*, e a sua linguagem deve ser explicitada via `[-l LANG]`. Em alinhamento com sua natureza experimental, o *Witch Cooking* proporciona o código-fonte formatado resultante como saída, sem modificar o conteúdo original, efetivamente estabelecendo uma formatação segura.

4.3 Materiais

O *Witch Cooking* foi programado em Rust (RUST..., 2023), que é uma linguagem amplamente reconhecida na comunidade de programadores (STACK..., 2023). Dentre os motivos que levaram à escolha dessa linguagem para o desenvolvimento, destacam-se: (I) seu gerencia-

[1] Neste trabalho, são considerados convencionais os estilos de formatação abordados nas páginas *Indentation style* e *Programming style* da Wikipédia. (INDENTATION..., 2023; PROGRAMMING..., 2023)

mento de memória seguro[2] (SOFTWARE..., 2022, p. 3); (II) seu excelente desempenho, a par às linguagens C e C++; (III) seus recursos de linguagens modernas, incluindo (A) expressividade, (B) *zero-cost abstractions*, (C) correspondência de padrões, (D) *closures*, (E) macros, etc.; (IV) seu suporte a múltiplas plataformas; além de (V) seu ecossistema de desenvolvimento fértil, que oferece (A) uma ampla variedade de bibliotecas, (B) *Cargo* (CARGO..., 2023), o gerenciador de pacotes, além do (C) *rust-analyzer* (RUST-ANALYZER..., 2023), uma implementação do *Language Server Protocol* (LSP) para a linguagem.

Para atender a seus objetivos, o *Witch Cooking* foi desenvolvido com base na biblioteca *Tree-Sitter* (TREE-SITTER..., 2023), a qual: (I) suporta uma ampla variedade de linguagens de programação e formatos de arquivos, além de (II) possuir uma linguagem de consulta versátil e extensível que, dentre outras características, permite (A) a realização de buscas complexas na árvore de sintaxe analisada de um código-fonte, bem como (B) a adição de predicados personalizados.

Para escrever o código do *Witch Cooking*, foi utilizado o editor de texto *Neovim* (NEOVIM..., 2023), com configurações personalizadas (SILVA, 2023a). Essas configurações possibilitaram a integração com o *rust-analyzer*, o que resultou em melhorias significativas na experiência de programação. Dentre os plugins utilizados na configuração, destacam-se: (I) o *nvim-treesitter* (NVIM-TREESITTER..., 2023), que permitiu a integração com o *Tree-Sitter*; além do (II) *playground* (PLAYGROUND..., 2023), que disponibilizou ferramentas para uma melhor compreensão do *Tree-Sitter* e de sua linguagem de consulta.

4.4 Métodos

Embora o *Tree-Sitter* não tenha sido originalmente projetado para formatação de código-fonte, o autor deste trabalho identificou o potencial da biblioteca, particularmente em relação à sua linguagem de consulta, que — como já ressaltado — é extensível. Assim, para permitir a formatação instruída, o *Witch Cooking* baseou-se na extensão e disponibilização de predicados designados a executar operações específicas de formatação. Em outras palavras, a concepção do autor era que a formatação ocorresse conforme a aplicação dos predicados estendidos em padrões especificados no arquivo de instruções, escrito na linguagem de consulta do *Tree-Sitter*.

Devido à natureza prototípica do software desenvolvido, foi adotada uma abordagem multifacetada neste trabalho. Inicialmente, foi conduzida (I) uma pesquisa bibliográfica que abrangeu termos-chave como: (A) *"code formatting"*, (B) *"Tree-Sitter"*, (C) *"programming style"*, e (D) *"style guide"*. Complementarmente, foi feita (II) uma pesquisa documental, que fundamentou a compreensão das tecnologias e bibliotecas — em especial, o *Tree-Sitter* — utilizadas no desenvolvimento do *Witch Cooking*.

À medida que se aprofundou o conhecimento sobre o *Tree-Sitter*, foram incorporados

[2] Embora Rust não apresente *garbage collection* — o que contribui significativamente para o desempenho —, a linguagem emprega *features* e conceitos como *borrow checker*, *ownership* e *lifetime*, garantindo um gerenciamento de memória seguro. Isso é especialmente relevante em linguagens de alto desempenho que não possuem coleta de lixo (e.g., C e C++). Estudos estimam que, nesse contexto, pelo menos 65% das vulnerabilidades de segurança sejam causadas por falta de segurança no gerenciamento de memória (GAYNOR, 2020).

elementos de (III) uma pesquisa experimental com o objetivo de explorar a aplicação da biblioteca como base para o algoritmo de formatação de código-fonte. Similarmente, foram integrados elementos de (IV) um estudo de caso com o propósito de analisar a eficácia do software desenvolvido. Essa abordagem metodológica se caracteriza como predominantemente qualitativa, de natureza básica e orientada por objetivos exploratórios.

4.5 Fazendo Consultas

Como mencionado na seção 4.2, o *Witch Cooking* formata o código-fonte conforme as instruções presentes no arquivo submetido via `-q QUERY`, o qual é escrito na linguagem de consulta do *Tree-Sitter*. Devido à importância fundamental desempenhada por essa linguagem na ferramenta, é essencial possuir um conhecimento básico sobre o seu funcionamento. Portanto, nesta seção, será abordado superficialmente o funcionamento da linguagem de consulta do *Tree-Sitter*, permitindo assim uma melhor compreensão da base em que o *Witch Cooking* foi desenvolvido.

Como abordado no capítulo 3, a linguagem de consulta do *Tree-Sitter* foi originalmente concebida para buscar padrões na árvore sintática via: (I) definição de padrões de correspondência e (II) aplicação de predicados. Os predicados disponibilizados nativamente pelo *Tree-Sitter* são agnósticos quanto à linguagem de programação utilizada. Por outro lado, os padrões de correspondência variam de acordo com a linguagem devido às suas construções gramaticais específicas, que são posteriormente traduzidas como nós na árvore de sintaxe analisada.

Por exemplo, considere os trechos de código 1 e 2, escritos nas linguagens de programação C e Rust, respectivamente. Embora ambos desempenhem papéis idênticos — i.e., definem uma função que retorna a soma do parâmetro `x` com 2 —, suas árvores de sintaxe analisadas — apresentadas respectivamente nos trechos de código 3 e 4 — exibem diferenças significativas. Portanto, para buscar a correspondência do trecho em que o parâmetro *x* é definido em ambos os códigos (i.e., `int add_two(int x) {` e `fn add_two(x: i32) -> i32 {`), são necessárias consultas distintas, apresentadas pelos trechos de código 5 e 6.

Dado que os padrões correspondem a qualquer uma de suas ocorrências, é importante garantir maior especificidade em sua definição. Isso pode ser obtido por meio de: (I) especificação de campos, (II) especificação de nós anônimos e (III) aplicação de predicados. Para ilustrar, ao aplicar a consulta apresentada no trecho de código 5 ao seguinte código em linha `int add(int x, int y);`, obtêm-se duas correspondências, como destacadas no seguinte trecho: `int add(int x, int y);`. Logo, para corresponder somente ao parâmetro `x`, deve-se utilizar uma consulta mais elaborada, conforme exemplificada no trecho de código 7. Nesse trecho, é possível notar algumas diferenças em relação à sua versão original, que foi apresentada no trecho de código 5. (I) O nó `(identifier)` é capturado com o nome `id`, como destacado no seguinte trecho de código: `declarator: (identifier) @id`. Além disso, (II) o predicado `eq?` é aplicado para garantir que o conteúdo da captura `id` corresponda à string “x” o que, por consequência, impede a correspondência do parâmetro `y`.

Por fim, é importante ressaltar que a linguagem de consulta do *Tree-Sitter* disponibiliza recursos adicionais (TREE-SITTER..., 2023) que vão além do escopo desta seção. No entanto, as informações abordadas até então estabelecem uma base fundamental para compreender o funcionamento da linguagem de consulta e sua aplicação em conjunto com o *Witch Cooking* na formatação do código-fonte.

Trecho de Código 1 — Função *add_two* em C

```c
int add_two(int x) {
  return x + 2;
}
```

Fonte: o próprio autor.

Trecho de Código 2 — Função *add_two* em Rust

```rust
fn add_two(x: i32) -> i32 {
  x + 2
}
```

Fonte: o próprio autor.

Trecho de Código 3 — Árvore Sintática da Função *add_two* em C

```
function_definition
  type: primitive_type
  declarator: function_declarator
    declarator: identifier
    parameters: parameter_list
      parameter_declaration
        type: primitive_type
        declarator: identifier
  body: compound_statement
    return_statement
      binary_expression
        left: identifier
        right: number_literal
```

Fonte: o próprio autor.
Nota: Nós anônimos foram omitidos.

Trecho de Código 4 — Árvore Sintática da Função *add_two* em Rust

```
function_item
  name: identifier
  parameters: parameters
    parameter
      pattern: identifier
      type: primitive_type
  return_type: primitive_type
  body: block
    binary_expression
      left: identifier
      right: integer_literal
```

Fonte: o próprio autor.
Nota: Nós anônimos foram omitidos.

Trecho de Código 5 — Consulta Correspondendo a um Parâmetro de Função em C

```
(parameter_declaration
  type: (primitive_type)
  declarator: (identifier))
```

Fonte: o próprio autor.

Trecho de Código 6 — Consulta Correspondendo a um Parâmetro de Função em Rust

```
(parameter
  pattern: (identifier)
  type: (primitive_type))
```

Fonte: o próprio autor.

Trecho de Código 7 — Consulta Correspondendo a um Parâmetro *x* de uma Função em C

```
(parameter_declaration
  type: (primitive_type)
  declarator: (identifier) @id
  (#eq? @id "x"))
```

Fonte: o próprio autor.

4.6 O Algoritmo de Formatação

O *Witch Cooking* permite a formatação de código-fonte via seu algoritmo fundamentado no *Tree-Sitter* e sua linguagem de consulta. Esse algoritmo utiliza a biblioteca para gerar uma árvore de sintaxe a partir do código-fonte submetido pelo usuário. Adicionalmente, complementando os predicados *builtins* do *Tree-Sitter* — os quais foram concebidos para a busca de padrões correspondentes —, o algoritmo inclui predicados desenvolvidos para executar procedimentos variados de formatação. Consequentemente, ao utilizar esses predicados nos padrões da consulta, é possível instruir a formatação de nós específicos da árvore de sintaxe.

A operação do algoritmo de formatação divide-se nas etapas de: (I) inicialização, (II) formatação, e (III) retorno, como ilustrado na fig. 1. Durante a etapa de inicialização — onde estruturas são instanciadas —, primeiramente, (I) a árvore de sintaxe — derivada do código-fonte submetido para formatação — é analisada. Em seguida, (II) um objeto[3] de consulta é gerado a partir do arquivo contendo as instruções de formatação, escrito na linguagem de consulta do *Tree-Sitter*. Esse objeto: (A) contém informações sobre padrões, capturas, propriedades e predicados; e (B) é posteriormente utilizado para obter as correspondências dos seus padrões.

Além disso, (III) é instanciada uma estrutura editora que contém: (A) o texto do código-fonte em edição — originado do código submetido pelo usuário —, juntamente com (B) um vetor de edições. Essa estrutura é utilizada pelos predicados para realizar edições no código. Na sequência, (IV) é inicializado um objeto de configurações, que é futuramente empregado pelos predicados para obter e definir configurações em diferentes níveis de escopo, sejam eles globais — i.e., abrangendo toda a consulta — ou locais — i.e., restrito a um padrão. Finalizando a etapa de inicialização, (V) a estrutura de correspondências é derivada: (A) do objeto de consulta, juntamente com (B) a árvore analisada e (C) o texto do código-fonte submetido. Simplificadamente, essa estrutura pode ser descrita como um iterador que fornece nós que correspondem a capturas presentes em padrões específicos.

O processo efetivo de formatação de código-fonte inicia na segunda etapa. Nessa fase, a dinâmica se baseia em (I) um *loop* que itera sobre todos os padrões especificados no arquivo de instruções de formatação. Em cada iteração, inicialmente, (II) é checado se o padrão é enraizado[4] ou não, determinando assim se o escopo de atuação do padrão é local ou global, respectivamente. Em seguida, (III) um segundo *loop* percorre as coleções de nós capturados para cada correspondência do padrão em questão. No corpo dessa estrutura de repetição, (IV) são aplicadas as configurações especificadas no padrão via o predicado *builtin* `set!`. Além disso, (V) são aplicados os predicados gerais — i.e., predicados não *builtins* — especificados no padrão em iteração. Pressupõe-se que esses predicados estejam dentre os providos pelo *Witch Cooking* para instruir a formatação.

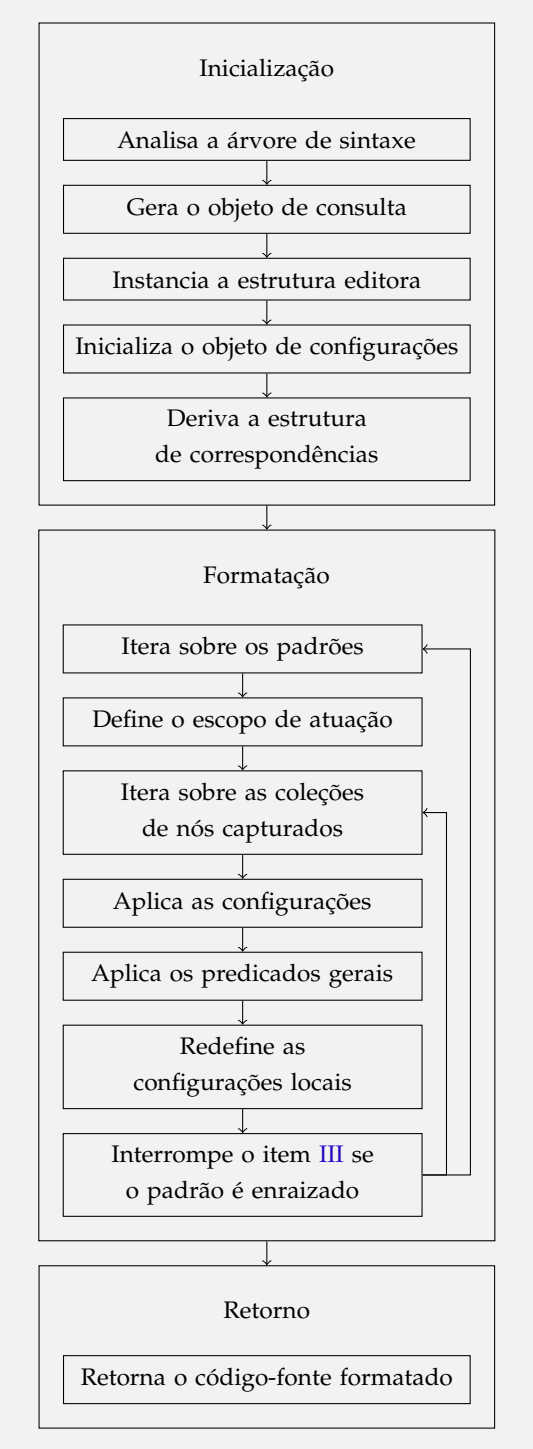
Encerrando o item III e, subsequentemente, o item I, bem como a etapa de formatação como um todo, (VI) as configurações locais — i.e, configurações a nível de correspondência —

[3] Neste contexto, o termo "objeto" é utilizado como uma alternativa para descrever uma estrutura em Rust.

[4] Na linguagem de consulta do *Tree-Sitter*, embora a biblioteca não forneça uma definição exata, um padrão enraizado é geralmente caracterizado por especificar pelo menos um nó inicial para a correspondência, e vice-versa.

são redefinidas para seus valores padrões. Logo em seguida, (VII) é interrompido o item III caso o padrão em iteração não seja enraizado. Por fim, na etapa de retorno, o algoritmo conclui retornando o código-fonte resultante do processo de formatação.

Figura 1 — Fluxograma do Algoritmo de Formatação



Fonte: o próprio autor.

4.7 Instruindo a Formatação

Como já enfatizado em diversas ocasiões, as instruções de formatação do código-fonte são especificadas no arquivo escrito na linguagem de consulta do *Tree-Sitter*, o qual é submetido ao *Witch Cooking* via `-q QUERY`. Para realizar um procedimento de formatação, é necessário: (I) definir um padrão de correspondência que delimita o escopo de operação; (II) capturar nós que (A) sejam alvos do procedimento (i.e., serão formatados) ou (B) auxiliarão nas operações; (III) opcionalmente, aplicar configurações via o predicado `set!`; e, por fim, (IV) aplicar os predicados estendidos pelo *Witch Cooking*.

Uma das operações fundamentais de formatação consiste na inserção de espaçamento entre *tokens*. Nesse contexto, o *Witch Cooking* disponibiliza o predicado `space!`. Esse predicado funciona da seguinte forma `(#space! [sep [lower [upper]]] a b)`, onde (I) `a` e `b` são duas capturas que se deseja separar; (II) `sep` é uma string opcional que determina a sequência de caracteres utilizada para separar os nós capturados, sendo o caractere de espaço o valor padrão; e (III) `lower` e `upper` são inteiros não negativos opcionais que definem o intervalo inclusivo para a não operação do espaçamento, sendo que (A) se ambos forem especificados, o intervalo será $\left[\texttt{lower}, \texttt{upper}\right]$, caso contrário, (B) se somente `lower` for especificado, o intervalo resultante será $\left[1, \texttt{lower}\right]$, senão, (C) caso ambos forem ausentes, o intervalo de não operação é nulo. Além disso, (D) quando o intervalo de não operação é definido, a separação dos nós é evitada se a sequência de caracteres atual entre eles contiver uma quantidade de `sep` que esteja dentro do intervalo especificado.

Para ilustrar o funcionamento do predicado `space!`, considere o seguinte vetor na linguagem de programação Rust, apresentado no seguinte trecho de código em linha: `[0,1,  2,  3]`. Para formatar esse vetor, de modo que cada número após uma vírgula seja separado por um espaço, é possível usar a consulta exibida no trecho de código 8. Ao formatar o vetor com essa consulta, é obtido `[0, 1, 2, 3]`, conforme esperado. No padrão presente na consulta usada para formatar o vetor, o predicado `space!` é chamado com a configuração de argumentos mais simples, que consiste em especificar apenas as capturas que devem ser separadas.

Trecho de Código 8 — Consulta para Separar Elementos de um Vetor em Rust

```
1  (array_expression
2    "," @comma
3    . (_) @item
4    (#space! @comma @item))
```

Fonte: o próprio autor.

Outro exemplo típico de inserção de espaçamento usado na formatação de código-fonte é o espaçamento entre elementos pertencentes ao corpo de uma função. Nesse contexto, é comum seprar esses elementos com uma quebra de linha. Contudo, em certos casos, são utilizadas duas quebras de linha para destacar diferentes blocos de código, a fim de favorecer a compreensão do leitor. Para ilustrar, considere o trecho de código 9, que contém dois blocos dentro de uma

função. No primeiro bloco de código, é possível notar o arranjo da linha 3, que contém duas *let declarations*. Para formatar o corpo da função, desaglomerando as *let declarations* e mantendo o espaçamento que separa os blocos, é possível utilizar a consulta mostrada no trecho de código 10.

Trecho de Código 9 — Função Com Bloco Aglomerado em Rust

```
1 fn x_plus_y() -> u32 {
2   let x = 5; let y = 11;
3
4   x + y
5 }
```

Fonte: o próprio autor.

Trecho de Código 10 — Consulta para Separar Elementos do Corpo de uma Função em Rust

```
1 (function_item
2   body: (block (_) @item . (_) @next)
3   (#space! "\n" 2 @item @next))
```

Fonte: o próprio autor.

Essa consulta apresenta um padrão que aplica o predicado `space!` com uma configuração de argumentos mais elaborada. Primeiramente, em adição ao trecho de código 8, (I) é especificada a string `sep` com o valor `"\n"`, indicando que os itens do corpo da função serão separados por uma quebra de linha. Além disso, (II) é especificado o limite inferior `lower` como 2, resultando num intervalo de não aplicação de $[1, 2]$. Em termos mais simples, isso implica que itens já separados por uma ou duas quebras de linha não sofrerão alterações, o que, por sua vez, mantém a formatação dos blocos intacta.

Desta forma, ao aplicar a consulta do trecho de código 10 ao trecho de código 9, obtém-se o código-fonte formatado, conforme ilustrado no trecho de código 11. Embora o uso de quebras de linha para separar elementos do corpo da função esteja correto, observa-se um erro de indentação na linha 4. Isso ocorre porque o predicado `space!` lida com a separação dos itens de sintaxe, mas não controla a indentação deles. No entanto, para lidar com a indentação, é possível combinar os predicados `set!` e `indent!`, como demonstrado no trecho de código 12.

Trecho de Código 11 — Função Com Má Indentação em Rust

```
1 fn x_plus_y() -> u32 {
2   let x = 5;
3 let y = 11;
4
5   x + y
6 }
```

Fonte: o próprio autor.

Trecho de Código 12 — Consulta para Separar Elementos do Corpo de uma Função Com Indentação em Rust

```
1 (#set! indent-style "  ")
2
3 (function_item
4   body: (block (_) @item . (_) @next)
5   (#space! "\n" 2 @item @next)
6   (#set! @next indent-rule "+1")
7   (#indent! @next))
```

Fonte: o próprio autor.

Na primeira linha deste trecho, observa-se um padrão composto exclusivamente pelo predicado `set!`, no qual a propriedade `indent-style` é configurada para dois caracteres de espaço. Conforme discutido na seção 4.6, a ausência de um nó como base para correspondência classifica esse padrão como enraizado, implicando que a propriedade especificada terá um efeito no escopo global. Simplificadamente, o trecho `(#set! indent-style "  ")` estabelece que, a menos que a configuração `indent-style` seja sobreposta localmente, o

estilo de indentação será caracterizado como dois espaços.

Ainda na consulta ilustrada pelo trecho de código 12, no padrão enraizado pelo nó `function_item`, começado na linha 3, podem ser observadas duas diferenças em relação à versão anterior, exibida no trecho de código 10. Primeiramente, (I) na linha 6, a propriedade `indent-rule` é configurada com o valor `"+1"` para a captura `@next`. Essa configuração define que, ao indentar os nós capturados, deverá ser utilizado o estilo de indentação vigente em adição ao deslocamento dos nós pais[5]. Além disso, (II) na próxima linha, o predicado `indent!` é aplicado, tomando como argumento a captura `@next`. Esse predicado tem como função aplicar a indentação previamente configurada a cada nó capturado pelos argumentos passados.

Finalmente, ao aplicar uma consulta mais sofisticada — exemplificada no trecho de código 12 — ao trecho de código 9, obtém-se o código-fonte formatado conforme ilustrado no trecho de código 13. Nesse código, observa-se que: (I) os blocos de código foram preservados, bem como (II) as *let declarations* foram desaglomeradas em suas respectivas linhas — e com a devida indentação —, atendendo às expectativas de formatação.

Trecho de Código 13 — Função Com Indentação Apropriada em Rust

```
1 fn x_plus_y() -> u32 {
2   let x = 5;
3   let y = 11;
4
5   x + y
6 }
```

Fonte: o próprio autor.

No decorrer da exemplificação do uso do predicado `space!` com a configuração de argumentos mais avançada, foi introduzida a configuração `indent-rule`. Essa configuração determina a regra de indentação para uma captura específica — i.e., a designação de uma captura é obrigatória para essa configuração. Através do predicado `set!`, é possível especificar os seguintes tipos valores para a configuração `indent-rule`.

Primeiramente, (I) pode ser definido um inteiro não negativo — sem sinal. Esse valor denota o nível de indentação absoluto para a captura em questão. Em outras palavras: (A) se o valor for 0, os nós capturados terão nenhum deslocamento em relação ao início da linha, independentemente do estilo de indentação definido via `indent-style`; (B) se o valor for 1, os nós capturados serão deslocados por uma string idêntica ao estilo de indentação definido; (C) se o valor for 2, os nós capturados serão deslocados por uma sequência de caracteres equivalente a duas strings subsequentes do estilo de indentação; e assim por diante.

Além disso, (II) pode ser especificado um inteiro positivo — com sinal. Esse número configura o nível de indentação adicional para os nós capturados, relativo ao deslocamento apresentado pelos pais desses nós. Dessa forma: (A) se o valor for `"+1"` [6], os nós capturados

[5] Tecnicamente, o deslocamento é calculado como o índice (em bytes) do primeiro caractere que não seja classificado como um *ASCII whitespace* (INFRA…, 2023) presente na linha que inicia o pai do nó capturado.

[6] É importante ressaltar que, ao fornecer um argumento para um predicado na linguagem de consulta do *Tree-Sitter*,

serão deslocados mais um `indent-style` além do que seus pais apresentam; já (B) se o valor for `"+2"`, o deslocamento adicional será de dois `indent-style`; e assim sucessivamente.

Encerrando, dentre os diferentes tipos de valores aceitos pela configuração de regra de indentação (`indent-rule`) via o predicado `set!`, (III) pode ser passado um inteiro negativo. Atualmente, esse valor é usado para calcular o número de caracteres pelo qual os nós capturados devem recuar em relação ao deslocamento dos seus pais. Esse cálculo é efetuado multiplicando-se o módulo do valor passado pela quantidade de caracteres que o estilo de indentação — definido via `indent-style` — contém. Por exemplo, considerando que o deslocamento dos nós pais é de 4 caracteres e que o estilo de indentação foi definido como `"  "` (dois espaços): (A) se -1 for passado, os nós capturados recuarão 2 caracteres e seu deslocamento será de 2 caracteres; (B) se o valor for -2, o recuo dos nós capturados será de 4 caracteres, e o deslocamento desses nós será de 0 caracteres — i.e., início da linha —; por fim, (C) se o número especificado for -3, ocorrerá um erro, pois o recuo calculado de 6 caracteres será maior que o deslocamento dos nós pais, que é de apenas 4 caracteres.

Outra prática comum de formatação consiste em alinhar a indentação de um nó com o deslocamento de outro nó arbitrário. Para alcançar esse tipo de formatação, pode-se fazer uso da configuração `indent-rule`. No entanto, devido à natureza dessa operação, que requer a relação de nós de capturas distintas, o uso do predicado `set!` não é adequado, uma vez que esse predicado permite apenas atribuir uma string a uma captura. Portanto, para estabelecer essa relação de deslocamento entre dois nós, o *Witch Cooking* disponibiliza o predicado `indent-offset!`. Esse predicado aceita duas capturas como argumentos, onde as regras de indentação dos nós capturados pela primeira captura serão definidas com base nos deslocamentos dos nós capturados pela segunda captura.

Para ilustrar o uso do predicado `indent-offset!`, tome como referência o seguinte trecho de código em linha: `fn foo() {bar()}`. Esse código, escrito na linguagem de programação Rust, consiste em uma função concisa. A fim de formatar essa função de acordo com a variante *1TBS* do estilo de indentação *K&R* (INDENTATION..., 2023), é possível utilizar a consulta mostrada no trecho de código 14. Nessa consulta, (I) estão presentes 3 capturas: (A) `item`, que captura qualquer elemento do corpo da função; (B) `close`, que captura o delimitador final do corpo da função; e (C) `fn`, que engloba toda a definição da função. Dentre essas capturas, (II) `item` tem sua regra de indentação especificada como um nível de indentação além do deslocamento do nó pai — i.e., a própria função —; e (III) `close` tem sua regra de indentação definida como o deslocamento de `fn` por meio do uso do predicado `indent-offset!`. Concluindo, ao final do segundo padrão, é utilizado o predicado `indent!`, passando as respectivas capturas `item` e `close` como argumentos. O resultado da aplicação dessa consulta está disponível para verificação no trecho de código 15.

esse argumento deve ser (I) ou uma string, (II) ou uma captura. Para especificar uma string, é possível seguir a abordagem tradicional, utilizando as aspas duplas como delimitadores para a sequência de caracteres. No entanto, quando cada caractere da string atende à expressão regular `[-0-9a-zA-Z]`, é possível dispensar os delimitadores. Consequentemente, para configurar uma regra de indentação, é possível especificar uma regra: (I) absoluta via `(#set! @node indent-rule 0)`, (II) relativa negativa usando `(#set! @node indent-rule -1)`, e (III) relativa positiva com `(#set! @node indent-rule "+1")`.

Trecho de Código 14 — Consulta para Formatar uma Função em Rust de Acordo Com o Estilo *1TBS*

```
(#set! indent-style "    ")

( (function_item
    body: (block (_) @item "}" @close)) @fn
  (#set! @item indent-rule "+1")
  (#indent-offset! @close @fn)
  (#indent! @item @close))
```

Fonte: o próprio autor.

Trecho de Código 15 — Função em Rust Indentada de Acordo Com o Estilo *1TBS*

```
fn foo() {
    bar()
}
```

Fonte: o próprio autor.

4.8 Falta Algo

Ao final do desenvolvimento do *Witch Cooking*, foi conduzida uma série de testes para verificar se o código formatado atendia às expectativas, conforme o código e as instruções submetidas. Um desses testes envolvia a formatação de duas funções aninhadas. O propósito desse teste era averiguar o comportamento da formatação em um contexto de aninhamento; ou seja, onde um trecho de código sofre o mesmo procedimento de formatação múltiplas vezes, mas com base em nós distintos.

Para realizar esse teste, foram submetidos: (I) o código-fonte escrito em Rust, que incluía as funções que configuravam o aninhamento, ilustrado no trecho de código 16; assim como (II) as instruções de formatação, escritas na linguagem de consulta do *Tree-Sitter*, apresentadas no trecho de código 17. A consulta submetida tinha o propósito de instruir a formatação, de modo que o resultado seguisse uma variante do estilo de indentação *1TBS* que utiliza 2 espaços em vez dos tradicionais 4 espaços. Assim, o código-fonte formatado deveria apresentar a estruturação apresentada no trecho de código 18.

Trecho de Código 16 — Funções Aninhadas em Rust

```
fn foo() {fn bar() {baz()}}
```

Fonte: o próprio autor.

Trecho de Código 17 — Consulta para Formatar Funções em Rust de Acordo Com uma Variante do Estilo *1TBS*

```
(#set! indent-style "  ")

( (function_item
    body: (block (_) @item "}" @close)) @fn
  (#set! @item indent-rule "+1")
  (#indent-offset! @close @fn)
  (#indent! @item @close))
```

Fonte: o próprio autor.

Trecho de Código 18 — Funções Aninhadas em Rust Conforme a Variante do Estilo *1TBS*

```
fn foo() {
  fn bar() {
    baz()
  }
}
```

Fonte: o próprio autor.

No entanto, ao analisar o resultado da execução do *Witch Cooking*, notou-se um código que destoa significativamente das expectativas, conforme apresentado no trecho de código 19. Esse código revela dois problemas relevantes. Primeiramente, é evidente que (I) a indentação

não está correta. Isso ocorre porque a expressão de função `baz()` possui o mesmo nível de indentação que a declaração de função `fn bar() {`, a qual engloba a expressão. Além disso, e de maior gravidade, percebe-se (II) a ausência de uma chave que caracterize o fechamento do corpo da função `fn foo() {`. Ou seja, a formatação de um código sintaticamente correto ocasionou um erro de sintaxe.

Trecho de Código 19 — Funções Aninhadas Mal Formatadas em Rust

```
1 fn foo() {
2   fn bar() {
3   baz()
4 }
```

Fonte: o próprio autor.

Para entender as causas de ambos os problemas, é necessário seguir uma explicação passo a passo dos procedimentos realizados. Começando pela má indentação presente na linha 3 do trecho de código 19, é essencial compreender o funcionamento do segundo padrão da consulta submetida, apresentado no trecho de código 17. Esse padrão, que lida com declarações de funções em Rust, (I) impõe um nível adicional de indentação em relação à função para os elementos dentro do corpo da própria função. Além disso, ele (II) alinha a chave de fechamento do corpo da função com o início da própria função.

Durante o processo de busca por correspondências do segundo padrão, realizado pelo *Tree-Sitter*, a primeira função a coincidir com o padrão é função `fn bar() {baz()}`. Como posteriormente, no algoritmo de formatação do *Witch Cooking*, a aplicação dos predicados ocorre sequencialmente conforme a ordem das correspondências, essa função é a primeira a ser formatada. Após a aplicação do primeiro procedimento de formatação, o código em formatação apresenta a configuração demonstrada no trecho de código 20.

Trecho de Código 20 — Primeiro Passo da Formatação das Funções Aninhadas em Rust

```
1 fn foo() {fn bar() {
2   baz()
3           }}
```

Fonte: o próprio autor.

Nesse código, destaca-se o deslocamento da expressão de função `baz()`, cuja regra de indentação foi especificada via `(#set! @item indent-rule "+1")`, conforme o trecho de código 17. Com base na explicação da seção 4.7, essa configuração define que o deslocamento da expressão de função, capturada via `@item`, será calculado como a combinação: (I) do estilo de indentação vigente com (II) o deslocamento do primeiro caractere não espaço encontrado na linha onde começa o nó pai da expressão. Nesse contexto, o nó pai é o corpo da função `bar`, que começa na linha 1. Dado que (I) o primeiro caractere não espaço nessa linha está no início da própria linha e (II) o estilo de indentação vigente é `"  "`, a indentação aplicada à expressão de função `baz()` configura uma formatação peculiar, como vista no trecho de código 20.

Prosseguindo com a compreensão dos procedimentos sucedidos, a segunda função a corresponder ao segundo padrão da consulta submetida é a função `foo`, que engloba todo o código. Similarmente ao ocorrido com a função `bar`, a aplicação do padrão nessa função supostamente resultaria na indentação: (I) da função `bar` e (II) da chave que fecha o corpo da função `foo`. No entanto, após a execução do segundo procedimento de formatação, o código problemático em estudo, apresentado anteriormente no trecho de código 19, é gerado.

Ao comparar o código resultante do segundo e último procedimento de formatação com o código esperado, ilustrado no trecho de código 18, observa-se que as duas primeiras linhas são idênticas, ou seja, seguem a variante do estilo *1TBS*. No entanto, na terceira linha do trecho de código 19, a indentação da expressão da função `baz()` é claramente inadequada, pois coincide com a indentação da declaração da função que a envolve. Esse problema decorre do cálculo de indentação executado pelo predicado `indent!`.

Conforme já explicado, na implementação atual, quando é estabelecida uma regra de indentação relativa positiva para um nó, o cálculo é baseado: (I) no deslocamento do primeiro caractere não espaço encontrado na linha do pai desse nó, e não (II) no deslocamento do próprio nó pai. No entanto, o cálculo com base no item II poderia ser mais apropriado, especialmente em circunstâncias de aninhamento. Por exemplo, se a indentação fosse calculada conforme o deslocamento do nó pai, ao realizar o primeiro procedimento de formatação no trecho de código 19, utilizando a mesma consulta do trecho de código 17 como base, o resultado seria conforme ilustrado no trecho de código 21.

Trecho de Código 21 — Primeiro Passo Hipotético da Formatação das Funções Aninhadas em Rust

```
1 fn foo() {fn bar() {
2                 baz()
3             }}
```

Fonte: o próprio autor.

Trecho de Código 22 — Segundo Passo Hipotético da Formatação das Funções Aninhadas em Rust

```
1 fn foo() {
2   fn bar() {
3     baz()
4   }
5 }
```

Fonte: o próprio autor.

Outro ponto passível de aprimoramento é a própria aplicação da indentação, que é também realizada pelo predicado `indent!`. Atualmente, esse predicado executa a indentação com base somente na linha inicial do nó em procedimento. Como resultado, ao aplicar a indentação à função `bar` no trecho de código 20, o conteúdo dessa função — a expressão de função `baz()` — que está em outra linha, permanece inalterado. Contudo, seria mais coerente se a indentação ocorresse em bloco, ou seja, o deslocamento seria aplicado para cada linha abrangida pelo nó em questão. Com essa melhoria, ao aplicar a consulta do trecho de código 17 ao trecho de código 21, o resultado seria o conteúdo ilustrado no trecho de código 22. Uma vez que esse código segue a variante do estilo *1TBS*, o teste teria seu objeto cumprido.

No entanto, ao retomar a análise do código problemático mostrado no trecho de código 19, o comparando com código hipotético resultante das melhorias discutidas, observa-se

que o código hipotético não apresentou um erro de sintaxe decorrente da falta da chave que fecha o corpo da função `foo`. Isso ocorreu porque, na exemplificação dessas possíveis melhorias, não foi considerado o problema que origina a omissão da chave em questão. Por sua vez, essa desconsideração ocorreu porque esse problema não deriva diretamente do *Witch Cooking*, mas sim de uma característica da biblioteca *Tree-Sitter*: ao sincronizar um nó, as edições ocorridas tangentes à sua extensão não são contabilizadas.

Para compreender a complexidade envolvida nesse problema, é essencial entender minimamente o funcionamento do processo de formatação; mais especificamente, como o processo de edição do código em formatação opera. Conforme mencionado na seção 4.6, o algoritmo de formatação é composto, entre outros componentes, por uma estrutura editora. Essa estrutura retém o código-fonte em formatação e registra as edições (ou modificações) realizadas pelos predicados.

Para realizar as edições, um predicado sempre se baseia em ao menos um nó. Dentre as informações que um nó possui, está a extensão de caracteres que o configura no código-fonte. Essa extensão compreende um início e um fim representados em ambos os níveis: (I) de bytes, especificando o deslocamento dentre os bytes do código-fonte; bem como (II) de pontos, onde cada ponto refere-se a uma linha e uma coluna específica. Originalmente, os nós capturados nos padrões mantém as extensões conforme presentes no código-fonte submetido. Portanto, ao manipulá-los, os predicados devem sincronizar os nós para que suas extensões estejam consoantes ao código vigente em processo de formatação.

Como demonstrado no apêndice A, ao sincronizar um nó, o *Tree-Sitter* não contabiliza edições tangentes à extensão desse nó. Consequentemente, especialmente em cenários que apresentam estruturas aninhadas, durante o processo de formatação, ao sincronizar um nó, há uma probabilidade considerável de que ele apresente uma extensão corrompida. Não surpreendentemente, essa é a razão por trás da omissão da chave, conforme demonstrada no trecho de código 19. Essa chave — que foi capturada via `@close` — tem sua regra de indentação definida como o deslocamento da função `foo`, como mostra o seguinte fragmento em linha do trecho de código 17: `(#indent-offset! @close @fn)`. Para indentá-la, mais especificamente, para aplicar os caracteres que configuram sua indentação, o predicado `indent!` busca um nó irmão que preceda a chave. Esse nó, por sua vez, é a função `bar`, a qual já fora modificada nos procedimentos anteriores. Como resultado, o predicado considera a extensão corrompida dessa função e, consequentemente, aplica a indentação que sobrescreve a chave que fecha o corpo da função `bar`.

4.9 Resultados

Conforme demonstrado ao longo desse capítulo, o algoritmo desenvolvido, que é fundamentado no *Tree-Sitter*, permite a formatação — multilíngue e personalizada — de código-fonte via as implementações que estendem a biblioteca. Mais especificamente, em adição à base fornecida pelo *Tree-Sitter*, a formatação é proporcionada pelas: (I) configurações, sendo elas (A) `indent-rule` e (B) `indent-style`; em conjunto com (II) os predicados, sendo eles

(A) `space!`, (B) `indent!`, e (C) `indent-offset!`.

Como explicado e demonstrado na seção 4.7, é possível desempenhar procedimentos básicos de formatação através do *Witch Cooking*. Nos exemplos utilizados, é perceptível que o código-fonte submetido para formatação já possui uma organização razoável. Ou seja, a ferramenta foi utilizada para realizar operações pontuais de formatação. Contudo, na seção 4.8, a partir de um cenário mais complexo de formatação, foi apresentado um mau funcionamento na sincronização das extensões dos nós. Conforme reportado, essa falha, que deriva de uma característica peculiar do *Tree-Sitter*, compromete o uso da ferramenta para efetuar procedimentos de formatação mais robustos. Portanto, ao considerar cenários mais realistas, nos quais não é possível presumir algo sobre a estruturação inicial do código, não é prudente pressupor que consultas mais complexas sejam capazes de abranger esses cenários sem ocasionar erros de sintaxe.

Um dos principais elementos no *prettyprinting* é o *CPL*, que se refere à quantidade máxima de caracteres permitida em uma única linha. No contexto da formatação de código-fonte, embora esse elemento tenha perdido parte de sua relevância atualmente[7], muitos guias de estilo para diferentes linguagens de programação ainda estipulam um valor para o *CPL* (CHARACTERS. . . , 2023). Consequentemente, o código formatado por diversos *prettyprinters* continua a ser diretamente influenciado por esse parâmetro. Apesar da importância que o *CPL* normalmente desempenha na formatação, o *Witch Cooking* atualmente não fornece algum mecanismo de formatação conforme esse aspecto. Mais precisamente, a ferramenta carece de mecanismos que proporcionem formatação condicionada, em geral.

[7] Embora o *CPL* tenha tido origem nas limitações técnicas dos equipamentos, as várias mudanças, incluindo melhorias significativas, no ambiente de desenvolvimento e nas tecnologias utilizadas transformaram o conceito de *CPL* em algo de menor relevância. Dentre os fatores que corroboraram para isso, destacam-se: (I) a disponibilidade de monitores de alta resolução, (II) editores de texto avançados, (III) recursos de formatação automática, e (IV) práticas de codificação modular.

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.

BLACK: The uncompromising Python code formatter. Versão 23.3.0. Black. Disponível em: <https://black.readthedocs.io/>. Acesso em: 8 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.

ECCLES, Stuart. **The Rise of Opinionated Software**. Medium. 28 mai. 2015. Disponível em: <https://medium.com/@stueccles/the-rise-of-opinionated-software-ca1ba0140d5b>. Acesso em: 26 mai. 2023.

FORD, Henry; CROWTHER, Samuel. **My Life and Work**. Edição: Samuel Crowther. [S.l.]: Doubleday, Page & Company, 1922. ISBN 1594621985.

FRIED, Jason; HANSSON, Heinemeier David; LINDERMAN, Matthew. **Getting Real**: The smarter, faster, easier way to build a successful web application. [S.l.]: 37signals, 18 nov. 2009. ISBN 0578012812. Disponível em: <https://basecamp.com/gettingreal>. Acesso em: 26 mai. 2023.

GAYNOR, Alex. **What science can tell us about C and C++'s security**. 27 mai. 2020. Disponível em: <https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/>. Acesso em: 26 set. 2023.

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.

INFRA Standard. Infra Standard. Disponível em: <https://infra.spec.whatwg.org/#ascii-whitespace>. Acesso em: 28 out. 2023.

JASPER, Daniel. **clang-format**: Automatic formatting for C++. Disponível em: <http://llvm.org/devmtg/2013-04/jasper-slides.pdf>. Acesso em: 27 mai. 2023.

LANCE, Corrina. **About opinionated software**: guidance VS freedom. Medium. 27 jul. 2021. Disponível em: <https://medium.com/codex/about-opinionated-software-guidance-vs-freedom-2bd66de304fe>. Acesso em: 27 mai. 2023.

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.

NATIONAL SECURITY AGENCY. **Software Memory Safety**: Cybersecurity Information Sheet. [S.l.], 22 nov. 2022. Disponível em: <https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF>. Acesso em: 29 set. 2023.

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.

PLAYGROUND: Treesitter playground integrated into Neovim. nvim-treesitter. Disponível em: <https://github.com/nvim-treesitter/playground>. Acesso em: 2 out. 2023.

OLIVEIRA, Delano and 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.

SCOWEN, Roger S. et al. SOAP: A Program which Documents and Edits ALGOL 60 Programs. **The Computer Journal**, v. 14, n. 2, p. 133–135, 1971. DOI: 10.1093/comjnl/14.2.133.

SILVA, Átila Gama. **UMA DLÇ**: A monkey-flavored configuration soup for Neovim. Disponível em: <https://github.com/atchim/uma-dlc>. Acesso em: 1 out. 2023.

_____. **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.

STACK Overflow Developer Survey. Stack Overflow. 2023. Disponível em: <https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages>. Acesso em: 26 set. 2023.

CARGO: The Rust package manager. Versão 1.73.0. The Rust Programming Language. Disponível em: <https://github.com/rust-lang/cargo>. Acesso em: 30 out. 2023.

RUST: A language empowering everyone to build reliable and efficient software. Versão 1.72.0. The Rust Programming Language. Disponível em: <https://www.rust-lang.org/>. Acesso em: 18 set. 2023.

RUST-ANALYZER: Bringing a great IDE experience to the Rust programming language. Versão 2023-09-25. The Rust Programming Language. Disponível em: <https://rust-analyzer.github.io/>. Acesso em: 1 out. 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.

TREE-SITTER: Using Parsers. Tree-Sitter. Disponível em: <https://tree-sitter.github.io/tree-sitter/using-parsers#pattern-matching-with-queries>. Acesso em: 22 set. 2023.

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

CHARACTERS per line. Wikipedia. Disponível em: <https://en.wikipedia.org/wiki/Characters_per_line>. Acesso em: 25 out. 2023.

INDENTATION style. Wikipedia. Disponível em: <https://en.wikipedia.org/wiki/Indentation_style>. Acesso em: 29 mai. 2023.

PROGRAMMING style. Wikipedia. Disponível em: <https://en.wikipedia.org/wiki/Programming_style>. Acesso em: 29 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

Characters per line (CPL) Na tipografia e na computação, *CPL* é um acrônimo para ***C**haracters **P**er **L**ine* — i.e., ***C**aracteres **P**or **L**inha* —, que se refere ao número máximo de caracteres que podem aparecer em uma linha de largura fixa. No passado, limitações na quantidade de caracteres por linha — geralmente de 70 a 80 — surgiram de restrições técnicas em equipamentos. Por exemplo: (I) os teletipos tinham entre 70 a 72 *CPL*; enquanto (II) as máquinas de escrever tinham cerca de 80 a 90; adicionalmente, (III) os tamanhos padrões de papel, como o A4, também impuseram restrições à largura da linha; além disso, na computação, (IV) cartões perfurados, assim como terminais populares, tinham 80 *CPL*, sendo que esse padrão ainda é comum em terminais virtuais atualmente.

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 *"**F**irst **I**n, **F**irst **O**ut"* — 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.

G

Garbage collection (GC) Em ciência da computação, *garbage collection* (GC), ou coleta de lixo, é uma forma de gerenciamento automático de memória. O coletor de lixo tenta recuperar a memória que foi alocada pelo programa, mas que não está mais sendo referenciada; essa memória é chamada de lixo. A coleta de lixo foi inventada pelo cientista da computação americano John McCarthy por volta de 1959 para simplificar o gerenciamento manual de memória em Lisp.

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**.

L

Loop Na programação de computadores, *loop* — também conhecido em português como "estrutura de repetição" ou "laço de repetição" — é uma sequência de declarações que é especificada uma vez, mas que pode ser executada várias vezes em sucessão. O código "dentro" do *loop* — o corpo do *loop* — é executado: (I) um número especificado de vezes, ou (II) uma vez para cada item de uma coleção, ou (III) até que alguma condição seja satisfeita, ou (IV) indefinidamente.

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. O termo foi popularizado a partir da obra *Getting Real: The smarter, faster, easier way to build a successful web application*, escrita por Fried, Hansson e Linderman (2009).

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*.

Pipeline Em sistemas operacionais de computador semelhantes ao Unix, *pipeline* é um mecanismo para comunicação entre processos usando passagem de mensagens. Um *pipeline* é um conjunto de processos encadeados por seus fluxos padrão, de modo que o texto de saída de cada processo (*stdout*) é passado diretamente como entrada (*stdin*) para o próximo. O segundo processo é iniciado enquanto o primeiro processo ainda está executando, e eles são executados simultaneamente.

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.

T

Token Na análise léxica, um *token* refere-se a uma unidade, que é a menor unidade reconhecível em um programa de computador escrito em uma linguagem de programação. Um *token* é uma sequência de caracteres que representa uma entidade específica, como uma palavra-chave, um identificador, um operador, um número ou um símbolo especial.

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; um (II) 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 em: (I) expressões regulares (*regex*), onde um caractere especial é usado para representar um conjunto de caracteres desconhecidos ou variáveis em uma string; e na (II) interface de linha de comando, onde as *shells* normalmente disponiblizam o caractere asterisco ('*') para corresponder a nomes de arquivos.

Z

Zero-cost abstractions Na programação de computadores, *Zero-cost abstractions* (ou abstrações de custo zero) é um conceito que se refere à capacidade de usar abstrações de alto nível e recursos da linguagem sem incorrer em penalidades de desempenho em tempo de execução. Isso significa que o programador pode escrever código de forma mais abstrata, expressiva e amigável para o desenvolvedor, enquanto o sistema subjacente gera código de máquina altamente eficiente.

Apêndices

Apêndice A — Sincronizando Nós do *Tree-Sitter*

Na seção 4.8, foi mencionado que, ao sincronizar um nó, a biblioteca *Tree-Sitter* não contabiliza as edições ocorridas que tangem a extensão desse nó. Este apêndice tem como objetivo demonstrar esse comportamento de forma didática. Para reproduzí-lo, é necessário: (I) Rust (RUST…, 2023), a linguagem de programação; (II) *Cargo* (CARGO…, 2023), o gerenciador de pacotes dessa linguagem; além das bibliotecas (III) *tree-sitter* e (IV) *tree-sitter-rust*.

Primeiramente, para criar um novo projeto, é possível executar o seguinte comando: `cargo new ts-node-edit`. Esse comando cria um novo pacote executável *Cargo*, nomeado de `ts-node-edit`, que será usado para realizar os procedimentos dessa demonstração. Em seguida, para navegar ao diretório recém-criado, com o mesmo nome do projeto, pode ser executado o seguinte comando: `cd ts-node-edit`. Dentro desse diretório, para adicionar as dependências — i.e., as bibliotecas — necessárias, é possível executar o seguinte comando em linha: `cargo add tree-sitter{,-rust}`. Após a execução desses comandos, a configuração inicial é concluída.

Para escrever o código, é necessário editar o arquivo localizado no seguinte caminho: `src/main.rs`. A seguir, serão mostrados e explicados os trechos de código que compõe esse arquivo. Essa abordagem fragmentada foi adotada por dois motivos principais: (I) explicar os procedimentos na ordem de execução, e (II) evitar problemas de formatação neste documento.

Conforme ilustrado no trecho de código 23, primeiramente, (I) são importados alguns itens da biblioteca *Tree-Sitter* que são utilizados nessa demonstração. Em seguida, (II) inicia-se a função principal, que engloba todos os procedimentos. Dentro dessa função, inicialmente, (III) é definido a string do código-fonte usado como referência na demonstração: `fn foo() {}`. Além disso, nas próximas linhas, (IV) são realizados procedimentos para gerar a árvore de sintaxe a partir da análise do código-fonte definido anteriormente.

Continuando, no trecho de código 24, observa-se que: primeiramente, (I) é obtido um nó que supostamente remete à função `foo`. Logo na sequência, (II) essa suposição é validada por uma asserção. Já nas linhas subsequentes, (III) são realizados procedimentos similares, porém buscando o nó que representa o nome da função. Em seguida, nos trechos de código 25 e 26, são executadas duas asserções semelhantes. Essas asserções verificam os valores originais das extensões — tanto em bytes quanto em pontos — dos nós: (I) da função `foo` e (II) do seu nome.

Uma vez que, antes de qualquer modificação ocorrer, foram avaliadas as extensões de ambos os nós, ou seja, (I) da função `foo` e (II) do seu nome, é possível mensurar o efeito da sincronização de uma edição para cada um deles. Para isso, é definida a edição ilustrada no trecho de código 27. Essa edição simula a inserção do caractere de quebra de linha imediatamente

após a palavra-chave `fn` em: `fn foo() {}`. É importante ressaltar que essa edição acontece: (I) antes do nome da função `foo`, bem como (II) dentro dessa função.

Prosseguindo, conforme o trecho de código 28, o nó que representa nome da função `foo` é sincronizado com a edição que simula a inserção da quebra de linha, mencionada anteriormente. Em seguida, ainda nesse trecho, uma asserção é executada para confirmar que a extensão desse nó refletiu essa edição. Já no trecho 29, finalizando o código, os mesmos procedimentos do trecho anterior são repetidos, com exceção de que: (I) o nó sincronizado dessa vez é o nó que representa a própria função e (II) a asserção feita verifica que esse nó não sofreu modificação em sua extensão.

Concluindo, para executar o projeto e constatar os resultados, é possível executar o comando: `cargo run`. Ao fazê-lo, observa-se que o código de execução 0 foi retornado, ou seja, a execução foi um sucesso. Isso implica que todas as asserções feitas no código eram verdadeiras. Por sua vez, as asserções feitas nos trechos de código 28 e 29 provaram que, com base na edição ilustrada no trecho de código 27: (I) ao sincronizar o nó representando o nome da função, localizado após a extensão da edição feita, o nó apresentou a extensão esperada, assim, refletindo a edição; já (II) ao sincronizar o nó remetendo à própria função, cuja extensão englobava a edição feita, esse nó não apresentou alguma modificação em sua extensão, fatalmente, não refletindo a edição feita.

Trecho de Código 23 — Demonstração da Sincronização de Nós do *Tree-Sitter* em Rust Parte 1

```
1  use tree_sitter::{InputEdit, Parser, Point, Range};
2
3  fn main() {
4    let src = "fn foo() {}";
5    let mut parser = Parser::new();
6    parser.set_language(tree_sitter_rust::language()).unwrap();
7    let tree = parser.parse(src, None).unwrap();
```

Fonte: o próprio autor.

Trecho de Código 24 — Demonstração da Sincronização de Nós do *Tree-Sitter* em Rust Parte 2

```
8    let mut fn_item = tree.root_node().child(0).unwrap();
9    assert_eq!("function_item", fn_item.kind());
10   let mut fn_name =
11     fn_item.child_by_field_name("name").unwrap();
12   assert_eq!("identifier", fn_name.kind());
```

Fonte: o próprio autor.

Trecho de Código 25 — Demonstração da Sincronização de Nós do *Tree-Sitter* em Rust Parte 3

```
13  assert_eq!(
14    Range {
15      start_byte: 0,
16      end_byte: 11,
17      start_point: Point { row: 0, column: 0 },
18      end_point: Point { row: 0, column: 11 },
19    },
20    fn_item.range(),
21  );
```

Fonte: o próprio autor.

Trecho de Código 26 — Demonstração da Sincronização de Nós do *Tree-Sitter* em Rust Parte 4

```
22  assert_eq!(
23    Range {
24      start_byte: 3,
25      end_byte: 6,
26      start_point: Point { row: 0, column: 3 },
27      end_point: Point { row: 0, column: 6 },
28    },
29    fn_name.range(),
30  );
```

Fonte: o próprio autor.

Trecho de Código 27 — Demonstração da Sincronização de Nós do *Tree-Sitter* em Rust Parte 5

```
31  let nl_edit = InputEdit {
32    start_byte: 2,
33    old_end_byte: 2,
34    new_end_byte: 3,
35    start_position: Point { row: 0, column: 2 },
36    old_end_position: Point { row: 0, column: 2 },
37    new_end_position: Point { row: 1, column: 0 },
38  };
```

Fonte: o próprio autor.

Trecho de Código 28 — Demonstração da Sincronização de Nós do *Tree-Sitter* em Rust Parte 6

```
39    fn_name.edit(&nl_edit);
40    assert_eq!(
41      Range {
42        start_byte: 4,
43        end_byte: 7,
44        start_point: Point { row: 1, column: 1 },
45        end_point: Point { row: 1, column: 4 },
46      },
47      fn_name.range(),
48    );
```

Fonte: o próprio autor.

Trecho de Código 29 — Demonstração da Sincronização de Nós do *Tree-Sitter* em Rust Parte 7

```
49    fn_item.edit(&nl_edit);
50    assert_eq!(
51      Range {
52        start_byte: 0,
53        end_byte: 11,
54        start_point: Point { row: 0, column: 0 },
55        end_point: Point { row: 0, column: 11 },
56      },
57      fn_item.range(),
58    );
59  }
```

Fonte: o próprio autor.