Witch Cooking

Formatação Multilíngue e Personalizada de Código-Fonte via o Sistema *Tree-Sitter*

Átila Gama Silva

1 de dezembro de 2023

INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Bahia
Campus
Irecê

Motivações

- As motivações surgiram das dificuldades ao estudar estilos de formatação de código-fonte em diversas linguagens de programação
  - Durante a análise dos estilos convencionais de formatação
    - Era imprescindível recorrer a diferentes *prettyprinters*
    - Cada um com suas próprias configurações e níveis de suporte para esses estilos
  - Durante a análise dos estilos não convencionais
    - A aplicação manual era inevitável, consumindo consideravelmente tempo e esforço

Problemática

- De modo geral, os formatadores de código-fonte
  - São restritos a uma linguagem específica ou a uma família de linguagens de programação
  - Oferecem uma quantidade limitada de configurações de estilização
  - Proporcionam pouca personalização

Objetivos Gerais

- Desenvolver um software de linha de comando — de natureza prototípica — para a formatação de código-fonte, tendo como objetivos
  - Abranger uma gama de linguagens de programação
  - Proporcionar a formatação personalizada via a linguagem de consulta do *Tree-Sitter*

Objetivos Específicos

- Desenvolver um algoritmo de formatação fundamentado no *Tree-Sitter*
- Definir configurações de estilização para o predicado `set!`, nativo da linguagem de consulta do *Tree-Sitter*
- Estender os predicados embutidos da linguagem de consulta, proporcionando predicados basais para a formatação

Resultados Esperados

- Que o *Witch Cooking* tenha potencial para abranger qualquer linguagem que tenha uma gramática gerada pelo *Tree-Sitter*
- Que os predicados desenvolvidos possibilitem a realização de procedimentos básicos de formatação

Limitações

- O desenvolvimento dos predicados foi limitado para fornecer o mínimo necessário para proporcionar procedimentos básicos de formatação
- Como resultado, não foram considerados mecanismos de formatação mais sofisticados, como a formatação condicional, no escopo deste projeto

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, 2015, p. 1)
- Esses métodos ganharam popularidade sob o termo *"prettyprinting"*
- No desenvolvimento de software, o *prettyprinting* é conhecido como formatação de código-fonte

A Formatação de Código-Fonte

- Durante as décadas de 60 e 70, a linguagem de programação LISP proporcionou condições favoráveis para o avanço da formatação de código (YELLAND, 2015, p. 2)
- Em 1967, Bill Gosper desenvolveu o *GRINDEF*: considerado o primeiro *prettyprinter* a mensurar o tamanho das linhas e ter ciência de sua localização no arquivo (GOSPER, 2023; GRIESEMER, 2022)
- Posteriormente, Oppen (1980) apresentou um algoritmo inovador capaz de formatar código-fonte derivado de qualquer linguagem de programação
  - O algoritmo necessitava que o código fosse anotado — por uma ferramenta intermediária — com espaços em branco e delimitadores especiais para marcar o início e fim de blocos

A Formatação de Código-Fonte

- Recentemente, Yelland (2015) descreveu um algoritmo que visa otimizar o layout do código em relação a uma noção intuitiva de custo de layout
  - Notavelmente, entre as abstrações de programação empregadas para facilitar sua aplicação em diversas linguagens e políticas de layout de código, destacam-se os *combinators*: funções geradoras que descrevem layouts alternativos para o código-fonte

O Sitema *Tree-Sitter*

- O *Tree-Sitter* (TREE-SITTER. . . , 2023) é um sistema multilíngue de análise sintática para ferramentas de programação, desenvolvido como uma tentativa de solucionar problemas presentes nas ferramentas de análise sintática da época, tendo como objetivos
  - Produzir árvores de sintaxe a partir da análise de códigos escritos em várias linguagens
  - Implementar a análise incremental, permitindo a atualização da árvore de sintaxe em tempo real
  - Expor através da árvore de sintaxe os nós representando suas construções gramaticais no código (e.g., classes, funções, declarações, etc.)
  - Ser livre de dependências, assim beneficiando sua adoção e aplicabilidade

O Sitema *Tree-Sitter*

- Além disso, o *Tree-Sitter* oferece uma pequena linguagem de consulta declarativa que é capaz de expressar padrões da árvore sintática por meio de *S-expressions* e buscar correspondências
- Essa linguagem suporta operadores que permitem
  - A captura de nós
  - A quantificação de nós, análoga às expressões regulares
  - O agrupamento de nós
  - As alternâncias de nós
  - O uso de *wildcards*
  - A ordenação de nós

O Sitema *Tree-Sitter*

- Adicionalmente, a linguagem de consulta permite 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

Materiais

- O *Witch Cooking* foi desenvolvido com base
  - No ecossistema Rust, composto
    - Pela linguagem de programação Rust (RUST..., 2023)
    - Pelo gerenciador de pacotes *Cargo* (CARGO..., 2023)
    - Pelo servidor de linguagem *rust-analyzer* (RUST-ANALYZER..., 2023)
  - No sistema/biblioteca *Tree-Sitter* (TREE-SITTER..., 2023)
  - No ecossistema Neovim, composto
    - Pelo editor de texto *Neovim* (NEOVIM..., 2023)
    - Pelas configurações personalizadas (SILVA, 2023)
    - Pelo plugin *nvim-treesitter* (NVIM-TREESITTER..., 2023)
    - Pelo plugin *playground* (PLAYGROUND..., 2023)

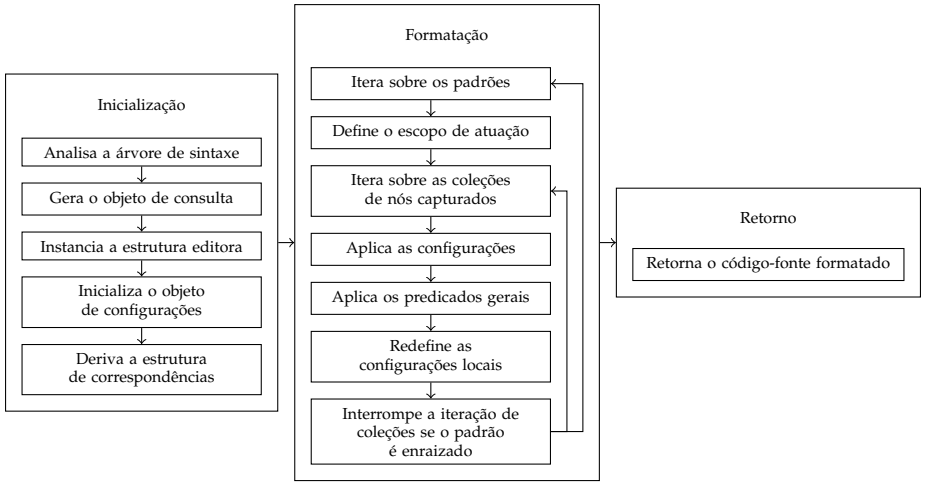
Métodos

- Foi realizado uma pesquisa experimental com o objetivo de explorar a aplicação do *Tree-Sitter* como base para o algoritmo de formatação de código-fonte
- Similarmente, foram integrados elementos de um estudo de caso com o propósito de analisar a eficácia do software desenvolvido

Usagem

- `cook [-l LANG] -q QUERY [SRC]`

O Algoritmo de Formatação



As Diretrizes de Formatação

- As diretrizes 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
  - Definir um padrão de correspondência que delimita o escopo de operação
  - Capturar nós que
    - Sejam alvos do procedimento
    - Auxiliarão nas operações
  - Opcionalmente, aplicar configurações via o predicado `set!`
  - Aplicar os predicados estendidos pelo *Witch Cooking*

As Configurações

- `indent-rule`
  - Define a regra de indentação para um nó
  - Seu valor pode ser
    - Um inteiro não negativo — sem sinal
    - Um inteiro positivo — com sinal
    - Um inteiro negativo
- `indent-style`
  - Define a string usada para indentar
  - Pode ser de escopo local ou global
  - Não aplicável a nós

Os Predicados

space!

- `(#space! [sep [lower [upper]]] a b)`
  - Separa os nós `a` e `b` com a string `sep`
  - Por padrão, `sep` é um caractere de espaço
  - `lower` e `upper` podem ser utilizados para definir um intervalo inclusivo para não atuação

Os Predicados

space!

Função Com Bloco Aglomerado em Rust

```rust
fn x_plus_y() -> u32 {
  let x = 5; let y = 11;

  x + y
}
```

Consulta para Separar Elementos do Corpo de uma Função em Rust

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

Função Com Elementos do Corpo Separados em Rust

```rust
fn x_plus_y() -> u32 {
  let x = 5;
let y = 11;

  x + y
}
```

Os Predicados

indent!

- `(#indent! node...)`
  - Aplica a indentação previamente configurada para cada `node`
  - Cada nó deve ter sua regra de indentação especificada — via `indent-rule` ou `indent-offset!`
  - Um estilo de indentação deve ser configurado — via `indent-style`

Os Predicados

indent!

Função Com Bloco Aglomerado em Rust

```rust
fn x_plus_y() -> u32 {
  let x = 5; let y = 11;

  x + y
}
```

Consulta para Separar Elementos do Corpo de uma Função Com Indentação em Rust

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

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

Função Com Indentação Apropriada em Rust

```rust
fn x_plus_y() -> u32 {
  let x = 5;
  let y = 11;

  x + y
}
```

Os Predicados

indent-offset!

- `(#indent-offset! target ref)`
  - Configura a regra de indentação — i.e., `indent-rule` — de `target` como o deslocamento de `ref`

Os Predicados

indent-offset!

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))
```

Função Compactada

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

Função em Rust Indentada de Acordo Com o Estilo *1TBS*

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

A Sincronização de Nós

- Durante a etapa de teste do funcionamento do *Witch Cooking*, notou-se uma peculiaridade — não documentada (DOCS. . . , 2023) — do *Tree-Sitter*
  - Ao sincronizar um nó, as edições ocorridas tangentes à sua extensão não são contabilizadas
    - Resultando em um potencial para erros de formatação

A Sincronização de Nós

Funções Aninhadas em Rust

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

Funções Aninhadas Mal Formatadas em Rust

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

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))
```

Conclusão

- Foi possível atender ao objetivo geral deste trabalho — desenvolver o *Witch Cooking* — a partir da implementação
  - Do algoritmo de formatação fundamentado no *Tree-Sitter*
  - Das configurações de estilização para o predicado `set!`
    - `indent-rule`
    - `indent-style`
  - Dos predicados basais para a formatação — estendendo a linguagem de consulta
    - `space!`
    - `indent!`
    - `indent-offset!`

Conclusão

- O software desenvolvido com base no sistema *Tree-Sitter* atendeu aos objetivos de
  - Abranger uma gama de linguagens de programação
    - Bash, C, C++, Go, HTML, Java, JavaScript, Python, Rust e TOML
    - A ferramenta tem potencial de suportar qualquer linguagem que tenha uma gramática compatível com a versão do *Tree-Sitter* utilizada no *Witch Cooking*
  - Proporcionar a formatação personalizada via a linguagem de consulta do sistema
    - É importante ressaltar que a ferramenta desenvolvida não é funcional para cenários realistas, uma vez que muito provavelmente ocasionará erros de sintaxe

Contribuições

- Diferentemente do algoritmo de Oppen (1980) que
  - Depende de um terceiro para fornecer o código com as anotações necessárias
  - Formata conforme o *CPL* em adição às anotações fornecidas
- O *Witch Cooking*
  - Alcança a formatação multilíngue através do *Tree-Sitter*
  - Proporciona predicados para dirigir a formatação
- Ao comparar essas duas abordagens, observa-se que
  - O *Witch Cooking* oferece maior controle ao usuário
  - Embora exija dele um conhecimento considerável
    - Da sintaxe em questão
    - Da linguagem de consulta

Contribuições

- Quanto às diretrizes de formatação, assemelham-se bastante ao que Yelland (2015) denomina de *combinators*
  - Funções geradoras para descrever layouts alternativos de código-fonte
- Mais precisamente
  - O predicado `space!` opera similarmente ao *combinator* $\leftrightarrow$
  - O predicado `indent!` funciona análogo ao uso de $\leftrightarrow$ em conjunto com o *combinator* $\updownarrow$
- Uma diferença significativa entre essas duas abordagens reside no fato de que
  - Os *combinators* se limitam ao algoritmo de formatação
  - Os predicados são disponibilizados ao usuário — por meio da linguagem de consulta do *Tree-Sitter* —, permitindo que ele componha suas próprias diretrizes de formatação

Trabalhos Futuros

- Desenvolver um algoritmo dedicado à sincronização de nós, capaz de sincronizar corretamente qualquer nó, mesmo após ele ter sofrido edições tangentes à sua extensão
- Aprimorar o predicado `indent!`
  - Melhorar a formatação de nós com regras de indentação relativa positiva, de modo que a indentação siga o deslocamento do nó pai, em vez de se basear no primeiro caractere não espaço encontrado na linha do pai
  - Seria mais coerente que esse predicado indentasse o bloco inteiro ao qual o nó se estende, em vez de se limitar à linha inicial do nó em questão

Trabalhos Futuros

- Aprimoramentos relacionados ao dinamismo da formatação
  - A formatação com base no *CPL*
  - A otimização para determinar o melhor layout
- Disponibilizar diretrizes de formatação que sigam os estilos de formatação convencionais para as linguagens de programação mais populares (a princípio)

Referências

- 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.
- 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.

Referências

- PLAYGROUND: Treesitter playground integrated into Neovim. nvim-treesitter. Disponível em: <https://github.com/nvim-treesitter/playground>. Acesso em: 2 out. 2023.
- 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. **UMA DLÇ**: A monkey-flavored configuration soup for Neovim. Disponível em: <https://github.com/atchim/uma-dlc>. Acesso em: 1 out. 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.

Referências

- DOCS.RS: Node in tree_sitter. Versão 0.20.10. The Rust Programming Language. Disponível em: <https://docs.rs/tree-sitter/0.20.10/tree_sitter/struct.Node.html#method.edit>. Acesso em: 5 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.

Referências

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.