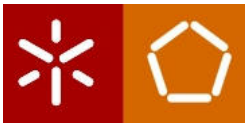

Universidade do Minho
Departamento de Informática

Perfil de Engenharia de Linguagens
Engenharia Gramatical

Trabalho Prático

Grupo 5

Gonçalo Costa - PG55944

José Correia - PG55967

Rodrigo Casal Novo - PG56006

Índice

1. Introdução

O grupo optou por escolher o **tema 1 - Beautifiers**. Beautifiers são ferramentas que formatam automaticamente o código-fonte, tornando-o mais legível e organizado conforme um conjunto de normas predefinidas. Estas ferramentas são amplamente utilizadas por programadores para assegurar que o código respeita convenções específicas, facilitando a manutenção e promovendo uma colaboração mais eficiente dentro das equipas de desenvolvimento.

1.1. Importância

- **Melhoria da legibilidade:** Um código formatado de forma consistente é mais fácil de compreender, reduzindo ambiguidades e facilitando a leitura.
- **Padronização do estilo:** Garante que todos os programadores seguem as mesmas regras de formatação, evitando discussões desnecessárias sobre estilo e promovendo a uniformidade do código.
- **Eficiência na escrita de código:** Automatiza a formatação, permitindo que os programadores foquem na lógica e na funcionalidade, em vez de perderem tempo com ajustes manuais.
- **Facilidade na revisão de código:** Quando a formatação está padronizada, as revisões de código podem focar-se na lógica e na qualidade do desenvolvimento, em vez de detalhes estéticos.
- **Integração com processos de CI/CD:** Muitas ferramentas de Beautification podem ser automatizadas dentro do ciclo de desenvolvimento, garantindo que o código é formatado corretamente antes de ser integrado no projeto.
- **Facilitação da Manutenção a Longo Prazo:** Projetos de software são continuamente atualizados e mantidos ao longo do tempo, muitas vezes por equipas diferentes. Um código bem formatado e padronizado torna futuras modificações mais fáceis, pois os programadores não precisam de perder tempo a tentar compreender um código confuso ou desalinhado.

2. Ferramentas

2.1. Prettier

O Prettier (Prettier Contributors 2025) garante que o código siga um estilo uniforme. Após analisar o código-fonte, reescreve-o de acordo com um conjunto de regras predefinidas. Por exemplo, remove espaços e quebras de linha desnecessárias, ajusta a indentação e padroniza o uso de aspas.

Um dos motivos da popularidade do Prettier é a sua facilidade de utilização e integração, podendo ser executado manualmente através da linha de comandos ou automaticamente integrado em IDEs como o VSCode, e além disso, pode ser configurado para ser aplicado antes de cada `push`, garantindo que todo o código submetido ao repositório segue um padrão uniforme. Hoje em dia a maior parte dos projetos Web já inclui o Prettier.

O Prettier pode ser combinado com o **ESLint**(ESLint Team 2025), onde o ESLint verifica o código em busca de erros e más práticas, o Prettier assegura a formatação correta, assegurando um código mais limpo e consistente.

Principais Linguagens: JavaScript, TypeScript, JSON, HTML, CSS, YAML, Markdown

2.2. Black

O Black(Python Software Foundation 2025) aplica uma formatação rígida e consistente ao código Python, o que é essencial para Python devido a sua sensibilidade com a indentação do código, estruturando automaticamente o código.

Assim como o Prettier, este pode ser executado através da linha de comandos ou configurado para ser aplicado automaticamente ao guardar um ficheiro na IDE do usuário, como é o caso do VSCode.

Principais Linguagens: Python

2.3. ClangFormat

O ClangFormat(LLVM Developers 2025) foca-se em linguagens compiladas, facilitando a revisão e manutenção do código.

Ao contrário das anteriores, o ClangFormat permite que os programadores definam um conjunto de regras de formatação num ficheiro de configuração (.clang-format). Ao executar a ferramenta, o código é reestruturado conforme essas regras. Pode ser integrado em IDEs como CLion e Visual Studio, aplicando automaticamente a formatação à medida que o código é escrito.(LLVM Developers 2025)

Principais Linguagens: C, C++, C#, Java, JavaScript

2.4. go fmt

O go fmt(Griesemer 2015) mantém um estilo uniforme de código em projetos escritos na linguagem Go, é uma ferramenta integrada no próprio ecossistema da linguagem. Ao executar o comando `go fmt`, o código é automaticamente formatado seguindo as diretrizes oficiais da linguagem, sem necessidade de configurações adicionais. Como resultado, todos os programadores que utilizam Go acabam por escrever código com um estilo homogéneo.

Principais Linguagens: Go

3. Comparação das ferramentes

Os Beautifiers empregam diversas abordagens técnicas para interpretar e modificar o código, que podem ser agrupadas em três categorias principais:

- Formatadores baseados em **AST** (Abstract Syntax Tree)
- Formatadores baseados em análise por regras (Fluxo de tokens e gramáticas)
- Formatadores integrados na linguagem

Cada uma dessas abordagens influencia diretamente a precisão, a flexibilidade e o desempenho da ferramenta.

3.1. Árvore de Sintaxe Abstrata (AST)

A Árvore de Sintaxe Abstrata (**AST**) é uma representação hierárquica da estrutura sintática do código, eliminando elementos desnecessários, como espaços e comentários. Utilizada internamente por compiladores e analisadores de código, permite interpretar e transformar programas de forma eficiente. Cada nó da árvore corresponde a uma construção sintática, representando expressões, declarações, operadores e blocos de código, ou seja, é possível entender a estrutura do código sem esta depender da formatação original. É fundamental para este tipo de ferramentas pois permite a formatação do código de forma segura e sem afetar a lógica.

A formatação baseada em **AST** segue um processo estruturado. Primeiro, o código-fonte é analisado e convertido numa **AST** através de um analisador sintático. Em seguida, essa estrutura é processada e formatada de acordo com as regras definidas pela ferramenta. Por fim, o código é regenerado a partir da **AST**, assegurando uma apresentação consistente e padronizada.

3.1.1. Exemplo prático de Black (Python)

Código-fonte:

```
def soma ( x, y ): return x+y
```

AST gerada:

```
Module(
    body=[
        FunctionDef(
            name='soma',
            args=arguments(
                args=[arg(arg='x'), arg(arg='y')],
            ),
            body=[Return(value=BinOp(left=Name(id='x'),op=Add(),right=Name(id='y')))]
        )
    ]
)
```

Novo código formatado:

```
def soma(x, y):
    return x + y
```

3.2. Formatadores Baseados em Parsing de Regras

Beautifiers como **Clang-Format** operam sobre o fluxo de tokens gerado pelo lexer ou parser, em vez de utilizarem diretamente a **AST**. Ao contrário da anterior, este utiliza a **AST** de forma implícita, isto é, a **AST** é gerada previamente pelo compilador do Clang.

Um Fluxo de Tokens é uma sequência de unidades lexicais ("tokens"), cada uma representando palavras-chave, identificadores, operadores, etc., sem formar uma estrutura hierárquica, mas mantendo a ordem e a relação entre os tokens. A formatação baseada nesta abordagem segue três etapas: primeiro, o código-fonte é convertido num fluxo de tokens; depois, são aplicadas regras de formatação para ajustar espaçamentos, indentação e quebras de linha; por fim, o código é reconstruído a partir dos tokens modificados, garantindo um estilo coerente.

3.2.1. Exemplo prático de Clang-Format (C++)

Código-fonte:

```
int main(){printf("Hello, World!");return 0;}
```

Fluxo de tokens gerado::

```
Keyword(int) Identifier(main) Punctuation(() Punctuation({)
Identifier(printf) Punctuation(() String("Hello, World!") Punctuation())
Punctuation(;)
Keyword(return) Number(0) Punctuation(;) Punctuation(})
```

Novo código formatado:

```
int main() {
    printf("Hello, World!");
    return 0;
}
```

3.3. Formatadores Embutidos na Linguagem

Um ótimo exemplo deste caso é a ferramenta **go fmt**(Mattermost 2020), pois não utiliza diretamente a **AST** nem opera apenas com tokens, mas baseia-se numa representação intermédia do código.

O Go usa go/scanner para dividir o código em tokens, go/parser para gerar a **AST** a partir desses tokens, e go/ast para representar essa árvore. Cada nó da **AST** corresponde a uma construção sintática do código, com informações de posição para rastrear onde cada elemento aparece no código-fonte. Em seguida, são realizadas verificações mínimas para garantir que a formatação não introduz erros. Por fim, o código é reescrito a partir da **AST**, seguindo um estilo único e predefinido, assegurando consistência e padronização.

3.4. Principais diferenças, vantagens e desvantagens

A principal diferença entre a **AST** e o **Formatadores Baseados em Parsing de Regras** está na forma como representam o código. A **AST** organiza os elementos de forma hierárquica, capturando a estrutura completa do programa, enquanto o fluxo de tokens é apenas uma sequência linear de unidades lexicais. A **AST** permite uma análise mais profunda do significado do código e possibilita a sua reescrita, enquanto a abordagem baseada em tokens limita-se a reorganizar a disposição dos elementos sem considerar a estrutura global.

A **formatação baseada em parsing de regras** oferece maior flexibilidade, permitindo personalizar o estilo do código conforme as preferências de cada pessoa. Além disso, é mais rápida do que a abordagem baseada em **AST**, pois não exige a reconstrução completa do código. No entanto, essa abordagem

apresenta algumas desvantagens: pode resultar em formatações inconsistentes, dependendo da configuração, e não garante a preservação total da semântica, já que atua diretamente sobre os tokens sem verificar a estrutura hierárquica.

Quanto às diferenças entre **Formatadores Embutidos** e de **ASTs**, ambas as ferramentas utilizam a **AST** para reescrever o código, mas com abordagens distintas. Por exemplo, o **Black** oferece algumas opções de personalização, como o limite de comprimento de linha, mas segue um estilo de formatação bem fixo. Já o **go fmt** segue um formato rígido, sem possibilidade de personalização, e utiliza a **AST** internamente para formatar o código, mas o utilizador não interage diretamente com ela.

3.4.1. Prettier vs Black

O **Prettier** não gera o código formatado diretamente a partir da **AST**, mas utiliza um algoritmo de impressão baseado na largura máxima da linha (**printWidth**). Para isso, divide o código em fragmentos e decide onde inserir as quebras de linha, ajustar espaçamentos e manter a legibilidade, sempre respeitando o limite definido.

Já o **Black** segue uma abordagem semelhante, utilizando um método chamado **pragmatic line wrapping** para organizar as quebras de linha conforme a largura máxima. No entanto, em vez de atuar diretamente sobre a **AST**, trabalha com tokens, o que permite preservar detalhes como comentários e strings formatadas.

Beautifiers	Linguagens	Técnica Base	Customização	Desempenho
Prettier	JavaScript, TypeScript, JSON, HTML, CSS	AST	Pouca	Lento por ser executado em Node.js
Black	Python	AST	Quase nenhuma	Rápido por ser compilado em C
ClangFormat	C, C++, C#, Java, JavaScript	LLVM / Fluxo de Tokens	Muita	Muito Rápido
go fmt	Go	Formatação da linguagem	Nenhuma	Muito Rápido

4. Casos de falha

Embora os Beautifiers sejam extremamente úteis para garantir a consistência e a legibilidade do código, eles podem falhar em situações específicas, como:

- **Falta de Flexibilidade:** Quando um código é altamente complexo ou utiliza convenções personalizadas, as ferramentas podem falhar porque a configuração predefinida não suporta aquela convenção, e mesmo beautifiers que utilizam um ficheiro de configuração, as regras de configuração podem não cobrir todos os cenários.
- ***Legacy Code*:** No caso em que o código foi alterado por diversas pessoas com estilos diferentes, aplicar uma destas ferramentas pode reformatar o código de forma inesperada.
- **Comentários e Strings:** Algumas ferramentas como o Prettier e o Black podem às vezes reformatar erradamente comentários ou strings que seguem um formato especifico.
- **Dependência da AST:** Quando o código contiver construções incomuns (como uso avançado de macros em C/C++) o **AST** gerado pode não refletir corretamente a estrutura do código, o que leva a formatação incorreta ou inconsistência na aplicação das regras. Este problema é ainda mais agravado em ferramentas não baseadas em **ASTs**, exatamente por não realizarem uma análise sintática e semântica tão profunda.

5. Conclusão

A análise dos Beautifiers evidenciou o papel central das ASTs e gramáticas na formatação de código. Ferramentas baseadas em ASTs, como Prettier e Black, garantem uma transformação estruturada e segura, enquanto abordagens que operam sobre tokens, como ClangFormat, oferecem maior flexibilidade, mas sem compreender a hierarquia sintática.

Essas diferenças ressaltam a importância de conceitos como análise sintática (Yacc, Lark) e lexers, fundamentais para a construção dessas ferramentas. Além disso, casos como go fmt mostram como a própria gramática de uma linguagem pode definir a formatação do código.

Assim, este estudo reforça a relevância da Engenharia Gramatical na criação de ferramentas que garantem padronização e manutenção eficiente do código-fonte.

Bibliografia

ESLint Team. 2025. «ESLint: Find and Fix Problems in Your JavaScript Code». https://eslint.org/.

Griesemer, Robert. 2015. «The Cultural Evolution of gofmt». https://go.dev/talks/2015/gofmt-en.slide#1.

LLVM Developers. 2025. «Clang: a C Language Family Frontend for LLVM». https://clang.llvm.org/.

Mattermost. 2020. «Instrumenting Go code via AST». https://medium.com/mattermost/instrumenting-go-code-via-ast-dc38addd25a5.

Prettier Contributors. 2025. «Prettier: An Opinionated Code Formatter». https://github.com/prettier/prettier.

Python Software Foundation. 2025. «Black: The Uncompromising Code Formatter». https://github.com/psf/black.