Universidade do Minho
Escola de Engenharia
Mestrado em Engenharia Informática

Unidade Curricular de Engenharia Gramatical 2025/2026

Beautifiers

Guilherme Pinto Pinho pg60263
Rodrigo Aires Lima Sousa pg60302

Janeiro, 2026

EG

Data da Receção	
Responsável	
Avaliação	
Observações	

Beautifiers

Guilherme Pinto Pinho pg60263
Rodrigo Aires Lima Sousa pg60302

Janeiro, 2026

Índice

Lista de Figuras

Lista de Tabelas

1. Introdução

A análise estática de código fonte é uma área fundamental da engenharia de software, permitindo extrair informação relevante sobre programas sem necessidade da sua execução. Estas técnicas são amplamente utilizadas em diferentes fases do desenvolvimento de software, nomeadamente para a deteção precoce de erros, verificação de propriedades semânticas, melhoria da legibilidade do código e apoio à sua manutenção e evolução.

O presente trabalho prático tem como objetivo o desenvolvimento de uma técnica de análise estática aplicada a uma linguagem imperativa simples, designada por IPL (Imperative Programming Language). Esta linguagem foi concebida especificamente no âmbito do projeto, de forma a suportar construções típicas de linguagens imperativas, como declarações de variáveis, estruturas condicionais, ciclos e funções. Para o seu processamento foi utilizada a biblioteca Lark, permitindo a construção da Árvore Sintática Abstrata (AST), a realização de análise semântica e a extração de representações intermédias, em particular o Control Flow Graph (CFG).

Como técnica de análise estática final, foi implementado um **Beautifier** para programas IPL. Esta ferramenta tem como principal objetivo melhorar a legibilidade e a organização do código fonte, através da aplicação sistemática de regras de formatação. A solução desenvolvida procura ainda dar resposta ao requisito de extensibilidade e personalização, permitindo ao utilizador adaptar aspetos como a indentação e a organização estrutural do código, sem alterar o seu comportamento semântico.

2. Descrição da Linguagem IPL

A linguagem IPL (Imperative Programming Language) foi concebida como uma linguagem imperativa simples, com especial enfoque na clareza sintática e na facilidade de aplicação de técnicas de análise estática. O seu principal objetivo é servir como uma base experimental para o estudo de parsing, análise semântica e construção de representações intermédias.

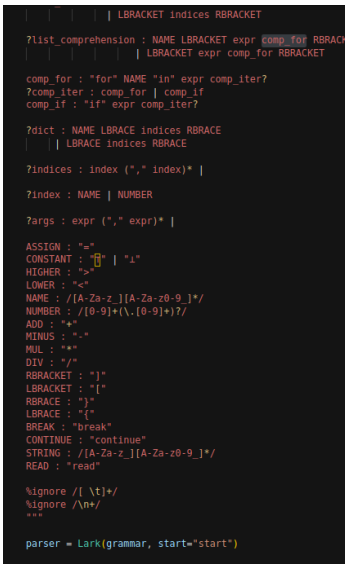
A linguagem suporta um conjunto representativo de construções comuns em linguagens imperativas, nomeadamente:

- Declaração de variáveis atómicas e compostas, através da palavra-chave *let*;
- Estruturas de controlo condicionais (*if*, *elif*, *else*);
- Estruturas de repetição, incluindo ciclos (*while* e *for*);
- Definição de funções sem parâmetros através da palavra-chave *func*;
- Estruturas de dados como listas, dicionários e listas por compreensão;
- Operações aritméticas e operadores de comparação;
- Mecanismos básicos de entrada e saída, através das instruções (*read* e *print*).

A gramática da linguagem foi especificada em EBNF e implementada utilizando a biblioteca Lark. De seguida, apresenta-se a gramática completa desenvolvida no âmbito deste trabalho.

```
from lark import Lark, Transformer

grammar = r"""
start : block

block : (stm)*

?stm : decl ";"
    | expr ";"
    | ifexpr
    | whilexpr
    | forexpr
    | funcdef
    | funcall
    | BREAK
    | CONTINUE
    | printexpr ";"
    | input_call ";"

printexpr : "print" "(" args ")"

?input_call : READ "(" ")"
            | "let" NAME ASSIGN READ "("")"

?decl : "let" NAME ASSIGN expr
      | "let" NAME LBRACKET indices RBRACKET
      | "let" NAME LBRACE indices RBRACE
      | "let" NAME ASSIGN funcall

ifexpr : "if" expr "{" block "}" elsexpr?
elsexpr : "else" "{" block "}"

whilexpr : "while" expr "{" block "}"

forexpr : "for" expr "in" expr "{" block "}"

?expr : assign

?assign : cmp (ASSIGN cmp)*

?cmp : sum ((HIGHER | LOWER) sum)*

?sum : term ((MUL | DIV) term)*

?term : factor ((ADD | MINUS) factor)*

?factor : NUMBER
        | CONSTANT
        | NAME
        | "'"STRING"'"
        | list
        | dict
        | "(" expr ")"

funcall : NAME "(" args? ")"

funcdef : "func" NAME "("")" "{" block returnexpr "}"

returnexpr : "return" expr? ";"

?list : list_literal | list_comprehension

?list_literal : NAME LBRACKET indices RBRACKET
```

Figura 1: Gramática IPL_1



```
    | LBRACKET indices RBRACKET

?list_comprehension : NAME LBRACKET expr comp_for RBRACK
                    | LBRACKET expr comp_for RBRACKET

comp_for : "for" NAME "in" expr comp_iter?
?comp_iter : comp_for | comp_if
comp_if : "if" expr comp_iter?

?dict : NAME LBRACE indices RBRACE
      | LBRACE indices RBRACE

?indices : index ("," index)* |

?index : NAME | NUMBER

?args : expr ("," expr)* |

ASSIGN : "="
CONSTANT : "⊤" | "⊥"
HIGHER : ">"
LOWER : "<"
NAME : /[A-Za-z_][A-Za-z0-9_]*/
NUMBER : /[0-9]+(\.[0-9]+)?/
ADD : "+"
MINUS : "-"
MUL : "*"
DIV : "/"
RBRACKET : "]"
LBRACKET : "["
RBRACE : "}"
LBRACE : "{"
BREAK : "break"
CONTINUE : "continue"
STRING : /[A-Za-z_][A-Za-z0-9_]*/
READ : "read"

%ignore /[ \t]+/
%ignore /\n+/
"""

parser = Lark(grammar, start="start")
```

Figura 2: Gramática IPL_2

3. Avaliação da Qualidade da Gramática

A qualidade de uma gramática é um fator determinante para a robustez, legibilidade e extensibilidade de uma linguagem de programação. Uma gramática bem desenhada facilita o processo de parsing, reduz ambiguidades, melhora a manutenção do código e simplifica a implementação de análises estáticas posteriores. Nesta secção é apresentada uma avaliação crítica da gramática da linguagem IPL, tendo em conta os atributos e métricas estudados nas aulas teóricas.

3.1. Clareza e Legibilidade

A gramática IPL apresenta uma estrutura clara e organizada, refletindo diretamente os principais conceitos de uma linguagem imperativa. As regras estão devidamente separadas por categorias semânticas (declarações, expressões, estruturas de controlo, funções), o que contribui para uma leitura intuitiva e facilita a compreensão global da linguagem.

A utilização de nomes de regras autoexplicativos, como ifexpr, whilexpr, forexpr, funcdef e printexpr, melhora significativamente a legibilidade da gramática, permitindo uma associação imediata entre a sintaxe e o seu significado semântico.

3.2. Expressividade e Cobertura da Linguagem

A gramática cobre um conjunto alargado de construções típicas de linguagens imperativas, nomeadamente:

- Declaração de variáveis atómicas e compostas (let);
- Estruturas condicionais (if e else);
- Ciclos (while e for);
- Definição e chamada de funções;
- Listas, dicionários e listas por compreensão;
- Operações aritméticas e comparações;
- Entrada e saída de dados (read, print);
- Definição de Comentários.

Esta variedade de construções garante uma boa expressividade da linguagem, permitindo escrever programas não triviais e servindo adequadamente como base para análises estáticas mais avançadas.

3.3. Estrutura das Expressões e Precedência de Operadores

A gramática define explicitamente a precedência e associatividade dos operadores através de uma hierarquia bem estruturada de regras (*assign, cmp, sum, term, factor*). Esta abordagem elimina ambiguidades

comuns em expressões aritméticas e relacionais, garantindo que a árvore sintática abstrata (AST) reflete corretamente a intenção do programador.

Por exemplo, a separação entre operações de comparação, soma/subtração e multiplicação/divisão assegura que expressões como x + y * z são interpretadas de forma correta e consistente.

3.4. Ausência de Ambiguidade

A gramática foi concebida de forma a minimizar ambiguidades sintáticas. A utilização de regras bem definidas para cada tipo de instrução e a clara separação entre declarações, atribuições e expressões contribuem para um parsing determinístico.

Além disso, a escolha do parser LALR reforça a necessidade de uma gramática não ambígua, requisito que é satisfeito na maioria das construções definidas. As alternativas opcionais são cuidadosamente controladas através do uso de operadores como ?, x e +, reduzindo conflitos de parsing.

3.5. Modularidade e Extensibilidade

A gramática IPL apresenta um bom nível de modularidade, uma vez que as diferentes construções sintáticas estão encapsuladas em regras independentes. Esta característica facilita a extensão futura da linguagem, permitindo a introdução de novas instruções ou expressões com impacto reduzido sobre as regras existentes.

Por exemplo, a adição de novos operadores, novos tipos de literais ou novas estruturas de controlo poderia ser realizada com modificações localizadas, sem necessidade de reestruturar a gramática global.

3.6. Complexidade e Manutenibilidade

Apesar da sua expressividade, a gramática mantém um nível de complexidade controlado, adequado aos objetivos do projeto. O número de regras e a profundidade das produções são suficientes para modelar uma linguagem imperativa realista, sem introduzir complexidade excessiva que dificulte a manutenção ou a análise semântica.

A utilização de EBNF contribui para uma definição compacta, reduzindo redundâncias e tornando a gramática mais fácil de evoluir.

3.7. Limitações Identificadas

Embora a gramática apresente uma boa qualidade geral, algumas limitações podem ser identificadas:

- A ausência de um sistema explícito de tipos na gramática limita a expressividade semântica da linguagem;
- As funções não suportam parâmetros, o que restringe a sua reutilização;
- A gestão de comentários é feita de forma simples, não sendo refletida na AST.

Estas limitações são aceitáveis no contexto do trabalho, uma vez que o foco principal está na análise estática e não na implementação de uma linguagem completa.

4. Implementação: parsing e AST

A gramática da linguagem IPL foi processada utilizando a biblioteca Lark, com o parser configurado no modo LALR, o que garante uma análise sintática eficiente, determinística e adequada a linguagens com gramáticas não ambíguas.

A partir do código fonte IPL, o parser gera automaticamente uma Árvore Sintática Abstrata (AST – Abstract Syntax Tree). Esta estrutura representa a organização hierárquica e lógica do programa, abstraindo detalhes puramente sintáticos como símbolos auxiliares, parênteses ou pontuação, e preservando apenas a informação relevante para as fases seguintes de análise.

Esta representação intermédia é fundamental, pois permite a aplicação de diferentes visitantes (Transformer, Visitor ou Interpreter) para realizar processamento semântico e estrutural sobre o programa.

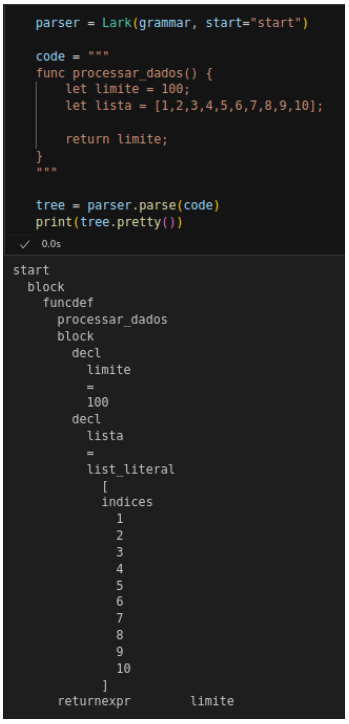


Figura 3: Árvore Sintática Abstrata (AST)

A AST serve de base para todas as fases subsequentes do projeto, nomeadamente:

- A análise semântica, incluindo verificação de escopos e tipos;
- A extração de representações intermédias, como o Control Flow Graph (CFG);
- A aplicação da técnica de análise estática final, neste caso um **Beautifier** configurável pelo utilizador.

5. Elaboração semântica

A fase de elaboração semântica tem como objetivo validar propriedades do programa que não podem ser verificadas apenas pela análise sintática. Para tal, foi implementado um analisador semântico recorrendo a um Transformer da biblioteca Lark, permitindo percorrer a Árvore Sintática Abstrata (AST) e aplicar regras semânticas de forma sistemática.

A análise de escopos é realizada através de uma pilha de dicionários, onde cada dicionário representa um escopo ativo. Sempre que é introduzido um novo bloco (por exemplo, numa estrutura condicional, ciclo ou função), é criado um novo escopo no topo da pilha, garantindo o correto isolamento de variáveis. No final do bloco, o escopo é removido, assegurando que apenas identificadores válidos permanecem acessíveis.

As principais verificações semânticas implementadas incluem:

- Deteção do uso de variáveis antes da sua declaração;
- Identificação de múltiplas declarações do mesmo identificador no mesmo escopo;
- Validação da existência de identificadores aquando da sua utilização.

Sempre que uma violação semântica é detetada, é lançada uma exceção do tipo SemanticError, interrompendo o processo de análise e fornecendo uma mensagem de erro clara ao utilizador. Este mecanismo contribui para a deteção precoce de erros lógicos no código fonte, aumentando a robustez da linguagem IPL.

Figura 4: Error - análise semântica

6. Extração de representação intermédia - CFG

O Control Flow Graph (CFG) é uma representação intermédia fundamental que descreve todos os possíveis caminhos de execução de um programa. Cada nó do grafo corresponde a um bloco básico de instruções, enquanto as arestas representam as transições de controlo entre esses blocos.

No âmbito deste projeto, o CFG foi construído através de um Transformer dedicado, que percorre a AST e gera progressivamente os nós e ligações correspondentes às estruturas de controlo do programa, como condicionais e ciclos. Esta abordagem garante que a estrutura lógica do programa é fielmente refletida no grafo resultante.

A biblioteca NetworkX foi utilizada para a representação interna do grafo, permitindo uma manipulação flexível da estrutura, enquanto a ferramenta Graphviz foi empregue para a sua visualização gráfica.

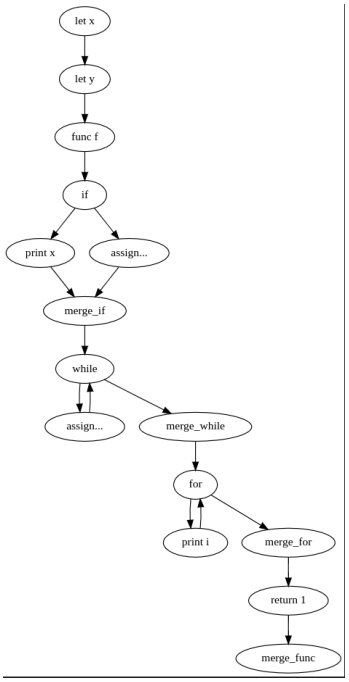


Figura 5: Grafo

A construção do CFG constitui um passo essencial para análises estáticas mais avançadas, como deteção de código morto, análise de caminhos de execução ou identificação de ciclos infinitos, servindo também como base para futuras extensões do projeto.

7. Análise Estática: Beautifier

Como técnica de análise estática final, foi desenvolvido um **Beautifier** para programas escritos em IPL. Esta ferramenta tem como principal objetivo melhorar a legibilidade, consistência e organização do código fonte, sem alterar o seu comportamento semântico.

O **Beautifier** atua diretamente sobre a estrutura sintática do programa, explorando a Árvore Sintática Abstrata (AST) para reconstruir o código de forma sistemática, de acordo com um conjunto de regras de formatação bem definidas. Desta forma, a técnica enquadra-se no domínio da análise estática, uma vez que opera exclusivamente sobre o código fonte, sem necessidade de execução do programa.

Entre as funcionalidades implementadas destacam-se:

- Indentação consistente e hierárquica de blocos de código;
- Separação clara e uniforme entre instruções;
- Reorganização estrutural de construções de controlo, como condicionais (if, else) e ciclos (while, for);
- Formatação consistente de expressões e chamadas de funções;
- Suporte à personalização por parte do utilizador, permitindo a definição de:
  - nível de indentação;
  - comprimento máximo das linhas;
  - espaçamento entre operadores;
  - estilo de utilização de aspas em literais.

Esta capacidade de configuração permite adaptar o estilo do código gerado às preferências do utilizador ou a convenções específicas de um projeto, tornando a ferramenta mais flexível e reutilizável.

A aplicação do beautifier contribui não só para uma melhoria visual do código, mas também para uma maior facilidade de leitura, compreensão e manutenção, especialmente em programas de maior dimensão ou complexidade.

📌 Exemplo de Beautification: **Antes:**

```
func processar_dados() {
let limite = 100;
let lista[1,2,3,4,5,6,7,8,9,10];

if limite>5{
                                    /*  é apenas um teste */
print([x+x for x in lista if x>limite]);
}else{
print(limite);
}
let ola = 0;
print('ola');
return limite;
}
```

Figura 6: Antes-Beautifier

Depois

```
func processar_dados(){
     let limite = 100;
     let lista [['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']];
     if limite     >     5 {
          /*  é apenas um teste */
          print([
           x     +     x
          for x
          in
          lista
          if x     >
          limite
          ]
          )
} else {
          print(limite)
}
     let ola = 0;
     print("ola")
return limite;
}
```

Figura 7: Depois-Beautifier

O Beautifier foi concebido de forma modular, permitindo a introdução de novas regras de formatação ou a adaptação das existentes sem alterações significativas à arquitetura do sistema. Esta abordagem garante o cumprimento do requisito de extensibilidade e personalização, conforme definido no enunciado do trabalho.

8. Discussões e Limitações

Apesar dos resultados obtidos, o projeto apresenta algumas limitações. Nem todas as construções da linguagem IPL são atualmente beautificadas de forma totalmente uniforme, em particular estruturas mais complexas ou menos frequentes, como listas por compreensão aninhadas ou expressões de maior profundidade sintática.

Adicionalmente, a análise semântica implementada é intencionalmente simples, focando-se sobretudo na verificação de escopos e no uso correto de variáveis. Funcionalidades mais avançadas, como verificação de tipos, suporte a funções com parâmetros ou validação mais rigorosa de expressões, não foram incluídas nesta versão do sistema.

Ainda assim, a arquitetura modular adotada ao longo do desenvolvimento facilita a evolução futura da ferramenta, permitindo a integração de novas funcionalidades e técnicas de análise estática com esforço reduzido. Esta modularidade garante que o projeto possa ser facilmente estendido, quer ao nível da linguagem IPL, quer ao nível das análises realizadas sobre o código.

9. Conclusão

Neste trabalho foi desenvolvida uma linguagem imperativa simples, bem como um conjunto integrado de técnicas de análise estática aplicadas à mesma. A utilização da biblioteca Lark permitiu implementar de forma eficaz as fases de parsing, análise semântica, construção de representações intermédias e aplicação de um beautifier configurável.

O projeto demonstrou a aplicabilidade prática dos conceitos estudados ao longo da unidade curricular, evidenciando a importância das representações intermédias, como a AST e o CFG, e das técnicas de análise estática na melhoria da qualidade, legibilidade e fiabilidade do código fonte.

Em particular, a implementação de um beautifier extensível e configurável permitiu cumprir os objetivos definidos para o trabalho, mostrando como a análise estática pode ser utilizada não apenas para deteção de erros, mas também como ferramenta de apoio à manutenção e evolução de programas.