Universidade do Minho
Escola de Engenharia

Manutenção e Evolução de Software

Trabalho Prático

MEI - 1º Ano - 2º Semestre

Trabalho realizado por:

pg57549 - Hugo Ricardo Macedo Gomes
pg57579 - Lara Beatriz Pinto Ferreira
pg57582 - Luís Miguel Moreira Ferreira

Braga, 27 de maio de 2025

Conteúdo

1. Introdução 3

2. Especificação da *Lang AST* 4

2.1. Exemplos de Uso 5

3. Gramática e *Parser* 6

4. *Pretty-Printing* 8

4.1. Propriedade de *roundtrip* 8

5. Otimizações e refatorações 9

5.1. Otimização 9

5.2. Refatoração 9

5.3. Análise estática: nomes, instruções e *smells* 9

5.4. Validação 10

6. Testes e avaliação 11

6.1. Avaliação de Programas 11

6.2. Testes unitários 11

6.3. Testes com Mutação 11

6.4. Instrumentação 12

6.5. Avaliação 12

7. Spectrum-Based Fault Localization 13

8. Conclusão e Trabalho Futuro 14

1. Introdução

Na sequência das aulas práticas de Manutenção e Evolução de *Software* foi-nos proposto o desenvolvimento de um projeto centrado na criação de uma linguagem de programação, na qual serão desempenhadas tarefas de análise, manutenção e evolução de *software*. Consequentemente, aplicou-se conceitos de *parsing*, *pretty-printing*, testes, *refatoring* e *SBFL* (*Spectrum Based Fault Localization*).

O grupo escolheu o desenvolvimento de uma linguagem estilo *Python* devido à sua legibilidade, familiaridade e ao uso de indentação como estrutura de blocos. As funcionalidades essenciais suportadas são: atribuições de variáveis, expressões aritméticas e booleanas; condicionais (*if--else*) e ciclos (*while* e *for*); definição e invocação de funções; por fim, insruções como *return* e *print*.

A representação abstrata dos programas (*AST*) foi implementada com recurso a *dataclasses* em *Python*, permitindo uma modelação clara e extensível de expressões, instruções e programas completos. Para a componente de *parsing*, recorreu-se à biblioteca *Lark*, utilizando uma definição gramatical *EBNF-like*. Com o suporte do *TreeIndenter*, foi possível preservar corretamente a estrutura dos blocos com base em indentação, refletindo a semântica de blocos da linguagem *Python*. A conversão da árvore sintática para *AST* foi feita através de um *Transformer*, responsável por mapear cada produção da gramática para a correspondente estrutura abstrata.

2. Especificação da *Lang AST*

Como já referenciado, o grupo decidiu conceber uma linguagem de programação simplificada inspirada na sintaxe e semântica da linguagem *Python*. A escolha justificou-se na legibilidade e famialiaridade da sintaxe, bem como pelo uso natural de indentação para estruturar blocos, o que facilita o *parsing* e a leitura dos programas.

A nossa linguagem suporta um conjunto essencial de instruções e expressões, definidas pelo enunciado, nomeadamente:

- Declarações e atribuições de variáves;
- Expressões aritméticas (+, -, *, /) e *booleanas* (&&, ||, ==, !=, <, >);
- Condicionais (*if … else*);
- Ciclos *while* e *for*;
- Definição e chamada de funções;
- Instruções especiais como *return* e *print*.

Para representar programas nesta linguagem de forma abstrata, foi definida uma árvore de sintaxe abstrata (*AST*) usando ***@dataclass*** em *Python*, o que permite uma modelação clara e extensível dos diferentes componentes da linguagem.

A hierarquia baseia-se nas seguintes categorias:

- ***Expr*:** representa todas as expressões (variáveis, literais, operações binárias e unárias, chamadas de função);
- ***Stmt*:** representa as instruções (atribuições, estruturas de controlo, declarações de funções, etc.);
- ***Program*:** representa a raiz do programa, como lista de instruções.

Um excerto da definição da *AST* é o seguinte:

```
@dataclass
class Var(Expr):
    name: str

@dataclass
class BinOp(Expr):
    op: str
    left: Expr
    right: Expr

@dataclass
class Assign(Stmt):
    var: str
    expr: Expr

@dataclass
class While(Stmt):
    condition: Expr
    body: List[Stmt]

@dataclass
class Program:
    body: List[Stmt]
```

2.1. Exemplos de Uso

Como forma de validar a expressividade e funcionalidade da *AST* desenvolvida, foram concebidos três programas *Lang* representativos — programa1, programa2 e programa3. Estes foram utilizados de forma transversal no projeto, servindo de base para testes de *parsing*, *pretty-printing*, execução, otimização, instrumentação e localização de falhas. Cada um cobre diferentes estruturas e padrões relevantes:

- **programa1:** soma decremental controlada por ciclo *while*, com condição composta e verificação de resultado.
- **programa2:** acumulação com *for* e operações redundantes, útil para refatoração e análise de código inútil.
- **programa3:** função com duplo condicional para validação de argumentos, enfatizando chamadas de função e escopos locais.

Estes exemplos cobrem todas as construções suportadas pela linguagem, e demonstram a versatilidade da *AST* tanto na sua definição como na manipulação posterior.

Exemplo do prorgama1:

```
programa1 = Program([
    Assign("y", IntLit(0)),
    Assign("temp", BinOp("+", IntLit(0), IntLit(3))),  # → 3
    Assign("useless", BinOp("*", IntLit(1), IntLit(1))),  # → 1
    While(
        condition=BinOp("==", BinOp(">", Var("x"), IntLit(0)), BoolLit(True)),  # →
apenas (x > 0)
        body=[
            Assign("y", BinOp("+", Var("y"), BinOp("*", Var("x"), IntLit(1)))),  # →
y + x
            Assign("x", BinOp("-", Var("x"), IntLit(0))),  # → x (não altera)
            Assign("x", BinOp("-", Var("x"), IntLit(1)))
        ]
    ),
    If(
        condition=BoolLit(True),  # → if sempre executado
        then_branch=[
            If(
                condition=BinOp("==", Var("y"), IntLit(6)),
                then_branch=[Assign("result", IntLit(1))],
                else_branch=[Assign("result", IntLit(0))]
            )
        ],
        else_branch=[
            Assign("result", IntLit(999))
        ]
    ),
    Return(Var("result"))
])
```

3. Gramática e *Parser*

Para interpretar programas escritos na linguagem *Lang*, foi desenvolvido um parser utilizando a biblioteca *Lark*, uma poderosa ferramenta de *parsing* em *Python* que suporta gramáticas escritas no estilo *EBNF* e inclui suporte nativo para blocos baseados em indentação, essencial para simular a semântica de blocos da linguagem *Python*.

A gramática da linguagem foi escrita de forma declarativa, abrangendo os elementos fundamentais da linguagem: atribuições, expressões aritméticas e booleanas, estruturas de controlo (*if*, *while*, *for*), funções e instruções *return* e *print*. A seguir, apresenta-se um excerto da gramática:

```
start: stmt*

stmt: assign_stmt
    | if_stmt
    | for_stmt
    | while_stmt
    | funcdef_stmt
    | return_stmt
    | print_stmt

assign_stmt: IDENTIFIER "=" expr _NL
return_stmt: "return" expr _NL

funcdef_stmt: "def" IDENTIFIER "(" parameters? ")" ":" _NL _INDENT stmt+ _DEDENT

if_stmt: "if" expr ":" _NL _INDENT stmt+ _DEDENT "else" ":" _NL _INDENT stmt+ _DEDENT

for_stmt: "for" IDENTIFIER "in" "range" "(" expr "," expr ")" ":" _NL _INDENT stmt+
_DEDENT

...

?expr: expr "==" expr   -> eq
        | expr "!=" expr   -> neq
        | expr "<" expr    -> lt
        | expr ">" expr    -> gt

...

_NL: (/\r?\n[\t ]*/ | SH_COMMENT)+

...
```

A biblioteca *Lark* permite o uso de *indenter* automático através do *TreeIndenter*, que trata a indentação como uma estrutura formal com tokens *INDENT* e *DEDENT*, permitindo representar blocos aninhados sem o uso de {}, como na linguagem C, ou *begin...end*, como em Pascal.

A conversão da árvore sintática concreta produzida pelo *parser* para a árvore de sintaxe abstrata (*AST*) foi realizada por uma classe ***ASTTransformer***, que herda de *lark.Transformer*. Cada método da classe corresponde a uma produção da gramática e é responsável por construir a respetiva instância da *AST*. Por exemplo:

```python
def assign_stmt(self, args):
    return Assign(str(args[0]), args[1])

def if_stmt(self, args):
```

```
    cond = args[0]
    stmts = self._filter_newlines(args[1:])
    half = len(stmts) // 2
    then_branch = stmts[:half]
    else_branch = stmts[half:]
    return If(cond, then_branch, else_branch)
```

O *parser* final pode ser utilizado de forma simples através da função ***parse_code(code: str) -> Program***, que transforma uma *string* com código *Lang* num programa representado como *AST*.

4. *Pretty-Printing*

Esta funcionalidade permite apresentar o conteúdo de um programa *Lang* de forma semelhante ao código-fonte original, usando uma sintaxe próxima de *Python*.

O módulo *pretty_printing.py* define funções recursivas que percorrem a *AST* e geram strings formatadas para cada tipo de expressão e instrução. Estas strings respeitam a indentação e a estrutura de blocos, essenciais para manter a legibilidade e a consistência do código gerado.

A função principal *pretty_program* produz o código textual completo a partir de um *Program*, e foi associada diretamente ao método *___str___()* da classe *Program*.

As funções *pretty_stmt* e *pretty_expr* geram as representações individuais de instruções e expressões, respetivamente.

4.1. Propriedade de *roundtrip*

Ou seja, a conversão de um *AST* para texto, seguida de *parsing* do mesmo texto, deve resultar num *AST* estruturalmente igual ao original. Esta propriedade foi testada em ***test_pretty_printer.py*** com vários exemplos de programas Lang. Os resultados da verificação da propriedade de *roundtrip* foram escritos no ficheiro ***resultado_roundtrip.txt***, o qual documenta para cada programa testado o código gerado pelo *pretty-printer*, a *AST* original e a *AST* obtida após *parsing*. Em caso de discrepância entre ambas as *ASTs*, é também incluída uma comparação detalhada linha a linha, permitindo identificar visualmente as diferenças estruturais através do uso de *difflib.unified_diff*.

A função ***prop_roundtrip(ast: Program) -> bool*** resume este comportamento e foi aplicada a todos os programas representativos desenvolvidos no projeto.

5. Otimizações e refatorações

Uma das componentes essenciais deste projeto foi a criação de um módulo de otimização e refatoração da linguagem *Lang*, cujo objetivo é melhorar a eficiência e legibilidade do código sem alterar o seu comportamento funcional. Para isso, foram desenvolvidas estratégias que percorrem a árvore de sintaxe abstrata (*AST*) dos programas e aplicam transformações locais a expressões e instruções.

5.1. Otimização

A função principal de otimização ***opt(prog: Program) -> Program*** aplica a estratégia ***simplify_stmt*** a cada instrução do programa, que por sua vez invoca ***simplify_expr*** para simplificar recursivamente expressões. As otimizações realizadas incluem:

- **Simplificações aritméticas:**
  - Eliminação de somas com zero;
  - Multiplicações por 0 ou 1;
  - Divisão por 1;
  - Constante folding;
- **Simplificações booleanas:**
  - true && expr → expr, false && expr → false;
  - true || expr → true, false || expr → expr
- **Eliminação de redundâncias triviais:**
  - Subtração de 0;
  - Divisão de 0.

5.2. Refatoração

A função ***refactor(prog: Program)*** aplica transformações que visam melhorar a legibilidade e reduzir *code smells*. As transformações incluem:

- **Simplificação de comparações *booleanas*:**
  - x == true → x;
  - x == false → not x.
- **Eliminação de condicionais triviais:**
  - if true: a else: b → a;
  - if false: a else: b → b.
- **Deteção de padrões redundantes, como:**
  - if com branches idênticos.

Estas estratégias foram implementadas de forma modular, permitindo aplicar otimizações e refatorações independentemente, mantendo o código original imutável.

5.3. Análise estática: nomes, instruções e *smells*

Foram também desenvolvidas funções de análise estática:

- ***names*:** recolhe os identificadores declarados num programa (variáveis e parâmetros de funções);
- ***instructions*:** contabiliza quantas vezes cada tipo de instrução (*Assign*, *If*, *Return*, etc.) aparece;
- ***detect_smells*:** deteta *code smells* simples, como:
  - Comparações redundantes com valores booleanos;
  - Branches idênticos em condicionais;

5.4. Validação

Todos os exemplos de programas definidos foram analisados automaticamente com o *script* ***test_optimization.py***. Este *script* executa as funções de otimização, refatoração, extração de nomes, contagem de instruções e deteção de *smells*, e guarda os resultados completos no ficheiro *resultado_optimization.txt*. A estrutura típica de saída inclui:

- *AST* original (via *pretty_printing*);
- *AST* otimizada;
- *AST* refatorada;
- Lista de nomes declarados;
- Frequência de instruções;
- *Smells* detetados.

Este processo automatizado permitiu validar a eficácia das transformações e destacar oportunidades de melhoria no código analisado.

6. Testes e avaliação

Para garantir o correto funcionamento dos programas escritos na linguagem *Lang*, foi desenvolvido um interpretador funcional que executa programas representados como *ASTs*, com base num ambiente de variáveis e funções

6.1. Avaliação de Programas

A função ***evaluate*** é responsável por interpretar um *Program*, dado um conjunto de *inputs* definidos como pares (nome_da_variável, valor_inteiro). Esta estrutura permite flexibilidade na avaliação de programas, podendo variar os contextos de execução sem alterar o código. Durante a execução, é mantido um dicionário *env* para variáveis globais e um ambiente *func_env* para definições de funções. A execução suporta:

- Atribuições e expressões aritméticas/booleanas;
- Estruturas de controlo: *if*, *while*, *for*;
- Chamada de funções definidas no próprio programa;
- Retorno de resultados através da instrução *return*;
- Impressão de valores com *print*.

6.2. Testes unitários

Foi desenvolvido um sistema de testes unitários parametrizados, que permite verificar se um programa *Lang* retorna o valor esperado para diferentes *inputs*. Para isso, são usadas as funções:

- ***runTest(programa, (inputs, expected)):*** verifica se o resultado de *evaluate* corresponde ao esperado;
- ***runTestSuite(programa, test_cases):*** aplica *runTest* a todos os casos de teste.

Três programas representativos (programa1, programa2, programa3) foram acompanhados dos seus respetivos ***testSuite***.

6.3. Testes com Mutação

Para avaliar a robustez dos testes unitários, foi implementado **Mutation Testing**. Esta técnica insere erros artificiais no código-fonte com o objetivo de verificar se os testes conseguem detetar esses erros — ou seja, se os testes falham quando o programa mutado se comporta de forma diferente do original.

A função ***mutate*** percorre a árvore sintática abstrata (*AST*) de um programa *Lang* e aplica uma mutação aleatória e significativa, selecionando um ponto válido do código. Entre as mutações implementadas, destacam-se:

- Alteração de operadores binários;
- Modificação de constantes inteiras;
- Transformações booleanas e aritméticas com impacto semântico;
- Filtragem de instruções irrelevantes: com a função ***collect_used_vars***, assegurou-se que mutações apenas são aplicadas a variáveis com impacto real na execução, evitando mutantes triviais como x = 0 nunca usados.

Estas mutações são aplicadas aleatoriamente com a função ***mutate***, que seleciona um nó mutável e aplica-lhe uma transformação via ***mutate_stmt***.

Cada programa representativo foi sujeito a mutações, sendo depois avaliado com a função ***runTestSuite*** contra os testes unitários definidos. O objetivo é que os testes falhem com o mutante, o que indica que são capazes de detetar comportamentos incorretos. Se o programa

mutado passar todos os testes, isso pode indicar um mutante equivalente (comportamento semântico idêntico ao original) ou um teste insuficiente.

6.4. Instrumentação

A função ***instrumentation*** insere instruções print especiais antes de cada instrução executável de um programa *Lang*. Cada print imprime um número inteiro único, que funciona como um identificador da instrução. Esta abordagem permite rastrear exatamente quais instruções foram executadas durante a avaliação de um programa, sem interferir na lógica original do código.

6.5. Avaliação

Todos os testes são executados via ***main*** no módulo ***test_evaluate.py***, e os resultados (incluindo execuções, mutações e *SBFL*) são escritos num ficheiro ***resultado_evaluate.txt***, que serve de *log* completo para avaliação.

7. Spectrum-Based Fault Localization

Com base nos identificadores recolhidos através da instrumentação, a função ***spectrum_based_fault_localization*** aplica uma métrica de suspeição para determinar quais instruções são mais prováveis de conter falhas. Neste projeto, utilizou-se a métrica ***Ochiai***, amplamente usada em *SBFL*, que calcula o score de cada instrução com base na sua frequência em testes que passaram e que falharam.

O processo é o seguinte:

1. Cada teste executado fornece um conjunto de Instr: N (instruções executadas).
2. Para cada instrução, conta-se:
   - Quantos testes falhados a executaram - fail(i);
   - Quantos testes passados a executaram - pass(i).
3. Aplica-se a fórmula:
   - Ochiai(i) = frac(fail(i), sqrt((fail(i) + pass(i)) * total_fail))

Além de validar a eficácia dos testes, a ferramenta de ***Spectrum-Based Fault Localization (SBFL)*** foi também aplicada aos programas mutados. Esta abordagem permitiu localizar as instruções responsáveis por falhas induzidas, ao associar a execução de instruções ao resultado dos testes (passaram ou falharam). O uso de ***SBFL*** revelou-se particularmente eficaz na deteção automática de pontos críticos do código, como condições mal formadas ou expressões aritméticas alteradas, destacando-as com altos scores de suspeição. Esta análise reforça a utilidade da instrumentação e do rastreamento de execução para apoio à manutenção e depuração de programas.

8. Conclusão e Trabalho Futuro

O desenvolvimento deste projeto permitiu aplicar diversos conceitos-chave da Unidade Curricular Manutenção e Evolução de *Software* sobre uma linguagem de programação criada de raiz. Através de uma abordagem prática e incremental, foram exploradas e implementadas funcionalidades fundamentais, tais como *parsing*, *pretty-printing*, otimização, refatoração, testes unitários e instrumentação.

A linguagem **Lang**, inspirada em *Python*, revelou-se uma escolha eficaz devido à sua legibilidade e estrutura por indentação, tendo facilitado o desenvolvimento da gramática, *parser* e *AST*.

As estratégias de otimização e refatoração aplicadas demonstraram ganhos em clareza e desempenho dos programas, com foco em simplificações aritméticas, booleanas e eliminação de redundâncias. Paralelamente, o sistema de ***code smells*** permitiu identificar padrões potencialmente problemáticos, apoiando a melhoria contínua da base de código.

No domínio da testagem, foi implementada uma infraestrutura completa de testes unitários, capaz de avaliar programas com base em inputs e resultados esperados. A incorporação de ***Spectrum-Based Fault Localization*** reforçou a capacidade de deteção de falhas, aferindo a cobertura e sensibilidade dos casos de teste.

Para o trabalho futuro, seria interessante explorar ***Property-Based Testing*** com ferramentas como Hypothesis, que permitiria testar propriedades gerais do comportamento dos programas, em vez de apenas casos específicos. Esta abordagem aumentaria significativamente a cobertura e robustez dos testes, ao gerar automaticamente inputs variados com base em invariantes específicas.

Outra possibilidade seria integrar ferramentas como ***mutmut***, um *mutation testing tool* em *Python*, para uma análise mais sistemática da eficácia dos casos de teste. A combinação entre mutações automatizadas e geração baseada em propriedades proporcionaria um ambiente de teste ainda mais rigoroso e sofisticado.

Globalmente, este projeto revelou-se uma experiência abrangente e desafiante, promovendo a aplicação prática de técnicas avançadas de manutenção de *software* e reforçando competências fundamentais na análise e evolução de código. O resultado final traduz-se numa linguagem funcional, extensível e bem instrumentada, suportada por um conjunto completo de ferramentas de apoio ao desenvolvimento e validação de programas.