Manutenção e Evolução de Software: Projeto Prático Versão 1

João Saraiva & José Nuno Macedo

Ano Lectivo 2024/2025

O projeto de Manutenção de Evolução de Software consiste na criação de uma linguagem de programação, na qual serão desempenhadas tarefas de manutenção e evolução. O projeto deverá ser desenvolvido na linguagem de programação `Python` e deverão ser guardados para fácil execução todos os resultados que considerar relevantes à avaliação em constantes com nomes relevantes, preferencialmente tudo num mesmo ficheiro separado. Por exemplo, se for pedida a escolha de 3 programas e eventualmente test suites à volta deles, dever-se-á definir:

```
programa1 = ...
programa2 = ...
programa3 = ...
(...)
testSuitePrograma1 = [(inputs1, 1), (inputs2, 17), (inputs3, -348)]
runTestSuitePrograma1 = runTestSuite programa1 testSuitePrograma1
```

Data de Entrega/Apresentação: TBD

1 Tarefas a Desenvolver

1.1 Language AST

Para esta primeira fase, deverá desenhar uma linguagem de programação - para o resto do enunciado, esta será chamada de linguagem `Lang`. Deverá desenhar a sintaxe da `Lang` como mais lhe agradar, p.e. poderá criar uma linguagem à la C ou à la Python, ou criar algo diferente se assim o desejar. A linguagem deverá conter as seguintes features:

- Declarações e usos de variáveis
- Expressões aritméticas e booleanas

- Expressões condicionais como por exemplo `if .. then .. else ..`
- Definição de loops, como por exemplo ciclos `for` e `while`
- (Extra) Definição e invocação de funções

Deverá criar a representação abstrata da sua linguagem de programação utilizando dataclasses, como foi exemplificado na aula em que foi utilizada a biblioteca pyZtrategic.

Para a validação da sua AST, deverá incluir 3 exemplos de programas válidos definidos usando a sintaxe que desenvolveu nesta fase.

Note-se que todas as fases seguintes do projeto podem ser executadas em paralelo desde que a AST do programa já tenha sido desenvolvida.

1.2 Program Parsing

Para a criação do parser da linguagem, deverá usar uma biblioteca de combinadores de parsing em Python - existem várias alternativas, entre as quais compynator, parsita, parsy e parsec.

Deverá certificar-se de que disponibiliza uma função `parse_code(code: str) -> Lang` que irá receber a representação textual de um programa e irá devolver a sua representação abstrata.

Para a validação do seu parser, disponibilize no seu projeto 3 exemplos de programas `Lang` válidos e 3 exemplos de programas inválidos em formato textual. O seu parser deverá produzir as respetivas ASTs para os programas válidos e rejeitar os programas inválidos.

1.3 Pretty-Printing

Nesta fase, deverá definir um pretty-printer para `Lang`, ou seja, software que pegue na representação abstrata de um `Lang` e produza a representação textual correspondente. O seu tipo de dados de entrada da linguagem deverá possuir o método `__str__()`, por forma que se possa fazer `str(prog)` para converter o programa `prog` na sua String correspondente.

Para a validação do seu pretty-printer, disponibilize no seu projeto 3 exemplos de programas `Lang` válidos em formato abstrato, que deverão ser convertidos para a sua representação textual. Atente também na seguinte propriedade:

```
def prop_roundtrip(ast: Lang) -> bool:
    return (parse (str(ast))) == ast
```

Disponibilize também a prova de que esta propriedade se verifica para os 3 exemplos de programas `Lang` que disponibilizou na fase de program parsing.

1.4 Optimization and Refactoring

Esta fase do projeto é dedicada à otimização e refactoring de programas `Lang`. Para isso, deverá usar a biblioteca pyZtrategic e definir estratégias que atravessem a árvore abstrata de um `Lang` apropriadamente.

1. Defina uma função `opt(ast: Lang) -> Lang` que deverá aplicar uma estratégia a um `Lang` que irá efetuar otimizações (não confundir com refactorings). Exemplos de otimizações:

   - Simplificações de somas com 0, multiplicações por 0 ou multiplicações por 1.
   - Resolução de operações aritméticas quando possível, p.e. soma de 2 valores constantes.
   - Simplificação de conjunções com valores True e de disjunções com valores False.
   - Simplificação de operações booleanas quando possível, p.e. `True && False || True`
   - Outras otimizações que considerar relevantes.

2. Defina uma função `refactor(ast: Lang) -> Lang` que deverá aplicar uma estratégia a um `Lang` que irá efetuar refactorings (não confundir com otimizações). Exemplos de refactorings:

   - Alteração de `if True then x else y` por apenas `y`.
   - Alteração de `x==True` por apenas `x`, e `x==False` por apenas `not(x)`.
   - Outros refactorings que considerar relevantes.

   Será valorizada a procura e implementação de refactorings relevantes em catálogos de code smells.

3. Defina uma função `names(ast: Lang) -> list[str]` que irá utilizar uma estratégia para recolher todos os nomes declarados em um `Lang`.

4. Defina uma função `instructions` que irá utilizar uma estratégia para contar quantas ocorrências de cada instrução acontecem num dado `Lang`.

5. Defina estratégias para a contagem de quantos/quais code smells ocorrem num dado `Lang`.

Para a validação das suas estratégias, deverá preparar exemplos de ASTs válidas de `Lang` e exemplos de utilização de todas as funções definidas sobre essas ASTs.

1.5 Software Testing

Nesta fase do projeto, vai-se implementar um sistema de Unit Testing sobre esta linguagem.

1. Defina uma função `evaluate(ast: Lang, inputs: Inputs) -> int` que, dado um programa `Lang` e os seus inputs, i.e. os valores atribuídos a cada variável, o execute e produza um resultado. Se necessário, adapte a sua linguagem para que um programa produzir um resultado (p.e. um inteiro) quando executado.

   Nota: Aqui, é usado o tipo Inputs para os inputs do programa. Caberá aos alunos adaptar isto conforme preferirem: podem, por exemplo, considerar que um input é um par que associe uma variável ao seu valor, e portanto definir `Inputs = List[Tuple[str, int]]`. Em alternativa, poderão usar este tipo diretamente na assinatura da função, ou qualquer outra solução que considerem adequada.

2. Defina uma função de execução de testes unitários `runTest(ast: Lang, testCase: Tuple[Inputs, int]) -> bool` que para um programa `Lang`, dado os seus inputs e o seu resultado esperado, irá verificar se o programa com esses inputs de facto produz esse resultado.

3. Defina uma função de execução de testes unitários `runTestSuite(ast: Lang, testCases: List[Tuple[Inputs, int]) -> bool` que irá correr vários testes unitários e validar se todos passam.

4. Selecione 3 programas `Lang` que considere representativos. Produza testes unitários para cada um, e verifique que estes passam com a sua função `runTestSuite`.

5. Desenvolva código para a inserção de uma mutação **aleatória** num programa `Lang`. Para isto, poderá recorrer a uma implementação manual, ou investigar as definições de `once_randomTP` e `mutations` disponíveis na biblioteca de programação estratégica em Haskell, adaptando conforme adequado para a sua utilização em Python.

6. Utilize o código da alínea anterior para inserir uma mutação em cada programa `Lang` escolhido anteriormente (ou insira manualmente se não o implementou). Utilize a função `runTestSuite` para correr os testes unitários definidos anteriormente, mas agora a usar o programa mutado (mantendo ainda o resultado esperado do programa original). Certifique-se que cada test suite irá agora falhar por causa da mutação feita.

7. Estenda a sua linguagem `Lang` para incluir uma instrução `print`, que poderá imprimir uma string para o terminal. Adapte o seu `evaluate` para que este seja capaz de imprimir texto sempre que encontra uma instrução `print`.

8. Defina uma função `instrumentation(ast: Lang) -> Lang` que irá pegar num programa `Lang` e instrumentar o programa para auxiliar a localização de falhas.

9. Defina uma função `instrumentedTestSuite(ast: Lang, testCases: List[Tuple[Inputs, int]) -> bool` que irá instrumentar um programa antes de o executar, e de seguida irá correr os testes unitários e validar se todos passam. Em alternativa, implemente uma versão da função `evaluate` que durante a execução do programa e inputs dados como argumento, produz no fim a lista de *instruções* do programa usadas durante a sua execução.

Para a validação do código desenvolvido nesta fase do projeto, deverá preparar 3 programas. Para cada um, deverá mostrar exemplos de execução da função `evaluate`, e deverá preparar test cases que os programas deverão passar. Por fim, deverá efetuar mutações nesses 3 programas e mostrar o resultado da execução dos mesmos test cases nas versões modificadas dos programas.

2 Extras

O projeto prático poderá ser mais valorizado se desenvolver extras relevantes a este projeto. Algumas **sugestões**:

- Investigue o que é *Spectrum-Based Fault Localization*. Recolha o resultado da execução de `instrumentedTestSuite` nos seus testes unitários que usam o programa mutado e o resultado esperado não-mutado. Coloque estes resultados numa tabela / folha de cálculo e utilize o algoritmo de *Spectrum-Based Fault Localization* para localizar as instruções com mais probabilidade de erro em cada um dos 3 programas definidos na secção de Software Testing.

- Investigue o que é *Property-Based Testing*. Defina geradores de `Lang` (sem e com noção de programa válido) e propriedades relevantes sobre `Lang`, que irá testar com os seus geradores. Para isto, recomenda-se recorrer à biblioteca Hypothesis, que permite property-based testing em Python.

- Investigue o funcionamento da ferramenta *mutmut* para Python. Esta é uma ferramenta para mutation testing de test suites em Python, e que permite aumentar a qualidade de uma suite de testes verificando a sua capacidade de apanhar bugs que a ferramenta injeta no código. Defina uma ferramenta semelhante, para a sua linguagem `Lang`. Isto é, a sua ferramenta deverá injetar bugs aleatórios em um programa usando o código de inserção de mutações desenvolvido no projeto, de seguida correr os testes unitários existentes, e por fim indicar ao utilizador quais as mutações sobreviventes.

3 Material a entregar

O grupo de trabalho deverá entregar todo o código que seja relevante ao projeto, assim como os slides que usar na apresentação em formato PDF. Embora não estritamente necessário, recomenda-se a preparação de uma Makefile ou instruções de execução do software. O desenvolvimento de um relatório não é necessário, e não serão beneficiados ou penalizados pela entrega de um relatório.

Todo o material a entregar deve ser colocado num ficheiro .zip e enviado por e-mail para `saraiva@di.uminho.pt` e `jose.n.macedo@inesctec.pt`.

Language AST	10%
Program Parsing	20%
Pretty-Printing	10%
Refactoring	20%
Software Testing	20%
Extras	10%
Apresentação	10%