—

Anônimo

[1]Departamento de Computação – Universidade Federal de Ouro Preto (UFOP)
35400-000 – Ouro Preto – MG – Brazil

Abstract. —

Resumo. —

1. Introdução

Com o aumento de cursos destinados a programação de computadores, muitos professores passaram a usar juízes automáticos (ou corretores automáticos de código-fonte) para auxiliar na correção de exercícios e fornecer *feedback* mais imediato aos alunos. Imagine que você é um professor de uma disciplina de programação e está avaliando uma questão cujo enunciado é o seguinte:

"Você foi contratado pelo Ministério do Meio Ambiente para avaliar a meta de reflorestamento das regiões brasileiras e vai implementar um programa para ajudá-lo em suas análises. Para facilitar a coleta de dados, cada estado é dividido em microrregiões. Você recebe periodicamente um vetor de valores inteiros indicando a quantidade mínima de árvores nativas plantadas para cada estado, representando a meta de cada estado, e uma matriz de valores inteiros que mostra a quantidade de árvores plantadas em cada estado em cada microrregião, as linhas da matriz representam as microrregiões e as colunas os estados. As entradas do vetor e da matriz são feitas por meio das funções inputVetor e inputMatriz, respectivamente (definidas no livro texto da disciplina).

Seu programa calcula o total de árvores plantadas pelos estados e avalia se eles cumpriram com a meta (quantidade de árvores plantadas é igual ou superior à meta do estado), imprimindo no terminal os estados que não conseguiram cumprir a meta (os números dos estados começam de 1, embora os índices comecem de 0, então, índice 0 representa o estado 1, índice 1 representa o estado 2, e assim por diante). A relação entre o vetor e a matriz se dá pelos índices dos elementos do vetor e os índices de coluna da matriz."

E, ao abrir a solução submetida por um aluno, se depara com esse código:

```
def inputVetor():
  entrada = input("Informe as metas dos estados: ")
  return list(map(int, entrada.split(',')))

def inputMatriz():
  entrada = input("Informe o plantio de arvores: ")
  linhas = entrada.split(';')
  matriz = [list(map(int, linha.split(','))) for linha in
      linhas]
  return matriz
```

```
def main():
    print("Ministerio do Meio Ambiente")
    metas = inputVetor()
    plantio = inputMatriz()

    num_estados = len(metas)
    totais_plantio = [sum(linha[i] for linha in plantio)
        for i in range(num_estados)]

    for i in range(num_estados):
        if totais_plantio[i] < metas[i]:
            print(f"Estado {i+1}, meta = {metas[i]}, plantio =
                {totais_plantio[i]}")

if __name__ == "__main__":
    main()
```

Que embora correto e gere a resposta esperada, usa recursos da linguagem Python que ignoram os objetivos de aprendizagem pretendidos ou não foram apresentados na disciplina, como o uso das funções `map` e `sum`, *list comprehension* e o uso do atributo `__name__`. Para atender a necessidade de não apenas validar a resposta, mas também validar o código submetido pelo aluno, propomos uma biblioteca para análise de código multilinguagem baseada em Parsing Expression Grammars (PEGs)[Ford 2004] na linguagem Haskell, projetada para detectar o uso de construções avançadas ou não autorizadas pelos alunos, visando auxiliar professores a impor limites pedagógicos.

2. Biblioteca: escolher um nome

A biblioteca permite ao usuário definir PEGs e padrões, e oferece operações para processar outros tipos de arquivos usando essas definições. A PEG é usada para realizar a análise sintática do arquivo processado, gerando uma Árvore Abstrata de Síntaxe (AST) do arquivo. Com a AST gerada, é possível realizar casamento de padrões, capturar subárvores e reescrever trechos da árvore, fazendo, assim, alterações no texto original.

Para tentar restringir o uso de construções indevidas, foi escrita uma PEG que aceita um *subset* de Python, permitindo apenas o que foi apresentado ao longo da disciplina de Programação de Computadores, como: definições e chamadas de funções, os comandos para decisão (`if`, `elif`, `else`), para repetição (`while`, `for`), as duas formas de importação (`import X` e `from X import Y`), os operadores de atribuição, lógicos e aritméticos, vetores e matrizes. Porém, apenas isso não é suficiente para eliminar totalmente todas as construções não autorizadas, como usar uma função definida em alguma biblioteca para resolver o problema, em vez de desenvolver a própria solução. Na Seção 3 são apresentados dois estudos de caso: um trata de uma forma de resolver o problema apresentado anteriormente e o segundo, dada a capacidade da biblioteca para reescrever a AST de um programa, trata de sugestões para reescrita do código, com o intuito de mostrar ao aluno formas de simplificar e melhorar a estrutura algorítmica do programa.

3. Estudos de caso

Para avaliar e mostrar as capacidades da biblioteca, apresentamos a seguir dois estudos de caso: um envolve uma sugestão de reescrita do código, enquanto o outro trata da verificação da presença de construções não autorizados na solução de um aluno.

3.1. Validação da resposta do aluno

Considere uma questão que pede ao aluno para implementar um programa que calcule o fatorial de um número inteiro n digitado pelo usuário. Definimos como $n!$ (n fatorial) a multiplicação sucessiva de n por seus antecessores até chegar em 1. A equação do fatorial pode ser definida como $n! = n \times n-1 \times n-2 \times \ldots 3 \times 2 \times 1$. A solução esperada é que o aluno utilize um laço de repetição, como o *while*, para implementar a multiplicação sucessiva dos números, como no código apresentado a seguir:

```
n = int(input("Digite um numero: "))
fatorial = 1
contador = n
while (contador >= 1):
  fatorial = fatorial * contador
  contador = contador - 1
print(f"{n}! = {fatorial}")
```

Porém, dentro da biblioteca `math` de Python, existe a função `factorial`, que dado um número inteiro n, retorna o resultado de $n!$. Por isso, alguns alunos acabam importando a biblioteca e usando a função pronta, contornando o objetivo do exercício, que é praticar o laço de repetição. Dessa forma, o padrão apresentado a seguir é capaz de identificar a presença de uma chamada para a função.

$$\begin{array}{ll} pattern factorial\_call: & function\_call := (identifier := "math.factorial")@space"("@space\#v2: \\ pattern space: & space := * \end{array}$$

Onde:

- A palavra *pattern* inicializa a declaração de um padrão.
- *factorial_call* é o identificador do padrão.
- *function_call* indica o tipo do padrão, i.e., com quais comandos ele vai casar.
- *(identifier := "math.factorial")* significa que o nome da função deve ser *math.factorial*.
- *@space* faz referência ao padrão de nome *space*.
- *"("* indica que deve casar com a string "(".
- *#v2:expr_list?* significa que a lista de argumentos da função será armazenada na variável *v2*. O *?* indica que essa lista de argumentos é opcional.
- ϵ indica que não deve ter nada em sequência da chamada.

Assim, se o padrão casar, significa que o aluno está usando a função *factorial* da biblioteca *math* em vez de escrever o laço de repetição, contornando o objetivo original do exercício.

3.2. Reescrita de código

Considere agora o seguinte trecho de código:

```
if not a:
  print("A■condicao■'a'■e■falsa")
else:
  print("A■condicao■'a'■e■verdadeira")
```

Embora o código não apresente nenhum erro, ele pode ser refatorado, com o intuito de melhorar a estrutura e, consequentemente, entendimento do código, ao remover o *not* da condição do `if` e trocar os blocos de comando do `if` e do `else`, da seguinte forma:

```
if a:
  print("A■condicao■'a'■e■verdadeira")
else:
  print("A■condicao■'a'■e■falsa")
```

Ao identificar esse tipo de construção no código do aluno, é possível sugerir uma reescrita ao aluno, explicando o motivo da sugestão e a melhoria que ela traria ao código. Os padrões apresentados a seguir representam uma forma de detectar a construção apresentada anteriormente e como fazer a reescrita.

$$
\begin{array}{ll}
patternif\_def: & if\_stmt := (("if"@space@expr" : ")\#ifBlock : statement*)@elseBlock \\
patternelseBlock: & else\_block := ("else"@space" : ")\#elseBlock : statement* \\
patternsubst: & if\_stmt := (("if"@space\#condition : expression" : ")\#elseBlock : statemer \\
patternelseBlock2: & else\_block := ("else"@space" : ")\#ifBlock : statement* \\
patternexpr: & expression := @orExpr\epsilon \\
patternorExpr: & or\_expr := @andExpr\epsilon \\
patternandExpr: & and\_expr := "not"@space\#condition : comparison \\
patternspace: & space := *
\end{array}
$$

Onde:

- O padrão *if_def* casa quando encontrar um if que tem como condição uma expressão negada.
- O padrão *subst* representa a reescrita que será sugerida ao aluno.

As variáveis $\#condition : expression$, $\#ifBlock : statement*$ e $\#elseBlock : statement*$ no padrão *if_def* capturam, respectivamente, a expressão na condição do `if`, todo o bloco de comandos do `if` e todo o bloco de comandos do `else`. A forma com que essas variáveis aparecem no padrão *subst* indica como a reescrita será realizada. Nesse padrão, é possível ver que o *not* da condição não aparece mais, enquanto a posição das variáveis dos blocos foram trocadas. Assim, é possível usar o que foi capturado pelas variáveis no padrão *if_def* e colocar nos locais onde as variáveis aparecem no padrão *subst*. Por fim, apresentamos a reescrita ao aluno, junto da explicação, e fazemos a sugestão para melhoria de sua solução.

4. Trabalhos relacionados

Estudantes aprendem melhor vendo representações de um conceito, sejam textuais, visuais ou animadas. Os livros didáticos podem oferecer as representações textuais e algumas visuais, enquanto os softwares podem fornecer representações visuais e animadas. No entanto, apenas observar não é suficiente, os alunos devem ser capazes de interagir com o conceito de alguma forma e receber *feedback* para verificar sua compreensão do conceito [Rodger 2002]. Dessa forma, avaliar a corretude da solução do aluno e oferecer sugestões de melhoria podem contribuir significativamente no aprendizado.

[Pelz et al. 2012] apresenta um mecanismo de correção automática de programas com um processo de 4 etapas, em que é são feitas as verificações da sintaxe, da presença de comandos obrigatórios, da adequação da estrutura do programa do aluno e dos valores de saída do programa diante de um conjunto de testes. A partir das observações que fizeram, concluíram que o *feedback* sobre a falta de comandos obrigatórios no programa deve ser utilizado com cuidado, pois ao ser apresentado ao aluno sem nenhuma precaução é possível que este descubra a estrutura da solução do problema simplesmente pelas dicas do mecanismo de correção, sem mesmo refletir sobre como solucionar o problema. Além disso, o mecanismo de correção automática dificilmente seria útil para avaliação de exercícios de programação mais complexos do que aqueles apresentados nas primeiras semanas de aula, já que neste caso a quantidade de variáveis e a complexidade das estruturas dos programas inviabilizam a definição dos esquemas de correção da maneira como estão propostos neste artigo.

[Moreira and Favero 2009] cita diversos benefícios para o professor ao usar um ambiente para ensino de programação com *feedback* automático, como: menor esforço, uma vez que conta com o auxílio da ferramenta, melhor administração dos estudantes e de suas tarefas, melhor rastreamento individual dos estudantes, melhor qualidade de ensino, devido o maior tempo de prática, mais tempo para contato com os estudantes. A utilização de ambientes para avaliação automática de exercícios tem sido uma prática comum em universidades e auxiliam no cotidiano de uma disciplina. Uma vez que o aprendizado de programação é essencial na carreira do estudante de computação, é necessário que este consiga adquirir os fundamentos básicos, sendo capaz de avaliar, pensar logicamente e desenvolver seus próprios algoritmos. Por se tratar de uma disciplina em que a prática em laboratório é fundamental no desenvolvimento destas habilidades, é desejável que o professor esteja atento à sua evolução. Contudo, devido às turmas de programação geralmente serem compostas por muitos alunos e tempo reduzido de aulas, nem sempre isso é possível. Por isso, uma forma de avaliar e fornecer *feedback* aos alunos pode auxiliar no acompanhamento dessas habilidades.

As principais abordagens para avaliação automática de um exercício de programação são a análise dinâmica e a análise estática [de Oliveira and Oliveira 2015]. A análise dinâmica compara o resultado produzido ao executar o programa do aluno com um gabarito, chamado de solução esperada e geralmente produzida por um programa escrito pelo professor, para avaliar a corretude, funcionalidade e eficiência da solução em vários casos de teste. Caso aprovados, o exercício recebe nota máxima e, caso contrário, nota mínima para cada caso. A análise estática avalia a escrita do código-fonte, observando itens como erros de programação de ordem sintática como erros sintáticos, semânticos, estrutural e estilo de programação, além de detectar plágio. Assim, ela cos-

tuma contemplar o processo de construção do programa.

Neste trabalho, foi apresentado um mecanismo para análise estática do programa doaluno, com um foco em coibir o uso de estruturas e construções avançadas ou não autorizadas pelos alunos, além de fornecer dicas e sugestões para melhorar a estrutura algorítmica da solução desenvolvida. Utilizando diferentes gramáticas, é possível restringir os recursos aos que foram ensinados na disciplina, disponibilizando mais recursos à medida que o conteúdo avança ou o quando o professor julgar adequado. Com o uso dos padrões, é possível garantir que determinados comandos e estruturas estão sendo usadas ou não, por exemplo, para que o aluno use uma estrutura adequada ao problema e não tente burlar o objetivo da atividade. Além disso, é possível detectar e sugerir melhorias na estrutura do código, explicando seu motivo e benefícios.

Referências

de Oliveira, M. and Oliveira, E. (2015). Abordagens, práticas e desafios da avaliação automática de exercícios de programação. In *Anais do IV Workshop de Desafios da Computação aplicada à Educação*, pages 131–140, Porto Alegre, RS, Brasil. SBC.

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122.

Moreira, M. P. and Favero, E. L. (2009). Um ambiente para ensino de programação com feedback automático de exercícios. In *Workshop sobre Educação em Computação (WEI 2009)*, volume 17.

Pelz, F., de Jesus, E., and Raabe, A. (2012). Um mecanismo para correção automática de exercícios práticos de programação introdutória. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, 23(1).

Rodger, S. H. (2002). Using hands-on visualizations to teach computer science from beginning courses to advanced courses. In *Second Program Visualization Workshop*, number 14, pages 103–112, Hornstrup Center, Dinamarca.