Perfil de EL - Engenharia de Linguagens (1º ano do MEI)

Trabalho Prático 3

Relatório de Desenvolvimento

Gonçalo Freitas (PG50398)

Henrique Alvelos (PG50414)

22 de abril de 2024

Resumo

Após no 2º Trabalho prático, estudarmos várias ferramentas para desenvolver um Analisador de código para uma evolução da Linguagem de Programação Imperativa (LPI), definida anteriormente por nós e, agora, passamos a desenvolver um criador de grafos: *Control Flow Graph* e *System Dependency Graph*

Conteúdo

Capítulo 1

Introdução

1.1 Enquadramento

Um Control Flow Graph (CFG) é uma representação gráfica do fluxo de controle dentro de um programa. Ele mostra as relações entre diferentes blocos de código e como a execução se move entre esses blocos. O CFG é composto por nós, que representam os blocos de código, e arestas, que indicam as transições de controle entre esses blocos. O CFG pode ser usado para analisar o fluxo de execução do programa, identificar caminhos críticos, encontrar loops e detectar possíveis problemas, como código inalcançável ou condições de corrida.

Já um System Dependency Graph (SDG) é uma representação gráfica das dependências entre diferentes componentes de um sistema. Ele mostra como os diferentes módulos, funções, bibliotecas e outros elementos do sistema dependem uns dos outros. O SDG é útil para entender as relações complexas entre os componentes e pode ajudar a identificar dependências circulares, avaliar o impacto de alterações em um componente específico e melhorar a modularidade e a manutenibilidade do sistema.

Ambos os grafos, CFG e SDG, são ferramentas poderosas para analisar e compreender o código-fonte de um programa. Eles podem ser usados em conjunto com um analisador de código estático para fornecer uma visão mais abrangente e detalhada do software. Com essas representações gráficas, os desenvolvedores podem tomar decisões mais informadas sobre a estrutura do código, identificar problemas potenciais e otimizar o desempenho e a manutenibilidade do sistema como um todo.

1.2 Problema

Foi proposto um enriquecimento, em Python, usando o Parser e os Visitors do módulo para geração de processadores de linguagens Lark.Interpreter, uma ferramenta que analise programas escritos na sua linguagem LPI e gere em formato *dot* os grafos standard de análise:

1. CFG (Control Flow Graph): Criação e representação para as seguintes instruções:
    - Estruturas cíclicas;
    - Instruções de declaração, atribuição e input/output.

2. SDG (System Dependecy Graph) “lite”, que apenas tem em consideração o controlo de fluxo, ignorando o fluxo dos dados.

1.3 Objetivo

Com este trabalho pretendemos aprender como produzir grafos com a linguagem que produzimos.

1.4 Estrutura do Relatório

No capítulo 2 são expostas as estruturas de dados e algoritmos utilizados para resolver o problema em questão.
No capítulo 4 termina-se o relatório com uma síntese do que foi dito, as conclusões e o trabalho futuro.

Capítulo 2

Concepção/desenho da Resolução

2.1 Estruturas de Dados

Tendo em consideração a opinião dada pelos professores na apresentação da segunda parte do trabalho prático e pela dificuldade em perceber o código que nós próprios fizemos, decidimos fazer um *refactor* sobre todo o trabalho. Para contextualizar, na parte anterior fazíamos uma pesquisa onde tudo era tratado nas folhas: guardávamos numa variável o código HTML, incrementávamos numa variável as estatísticas pretendidas e no fim devolvíamos um dicionário com todos os dados.

Assim, em vez de tratarmos tudo nas folhas, devolvemos dados aos nodos superiores. Estes dados são diferentes classes que criamos. Uma parte destas classes representam maioritariamente os símbolos não-terminais, isto é:

- Função
- Declaração
- Seleção
- Caso
- Repetição
- Atribuição

Todas estas classes têm, pelo menos, quatro métodos, universais entre elas:

- **html**();
- **___str___**();
- **cfg**();
- **sdg**().

Isto vai facilitar a chamada das funções para determinados objetivos, visto que não é preciso verificar o tipo das classes.

Quanto ao nosso interpreter, decidimos mudar algumas variáveis, ficando com:

Vars: Dicionário de Variáveis. Cada Variável contém um tipo, um id, um valor, o número de vezes que foi declarada e inicializada e, por fim, os seus atributos;

Funções: Dicionário de Funções. Cada Função contém um id, um dicionário de Variáveis que são os parâmetros, uma lista de instruções, uma expressão do retorno, um dicionário de variáveis criadas no corpo da função, o valor da Complexidade de McCabe e, por fim, uma lista com os nomes das funções cujo valor da Complexidade depende;

Instruções: Lista de Instruções do ficheiro;

Estatísticas: variável que contém vários atributos de modo a responder às tarefas pedidas anterior e atualmente;

Func_ID: Nome da função que está a ser processada. Caso não esteja a ser processada, esta variável é nula;

Erros: Lista com possíveis erros que devem ser detetados na pesquisa das folhas na árvore;

McCabeFunction: Valor atribuído pelo utilizador e representa a função que vai ser usada para calcular a complexidade de McCabe;

McCabeValue: Valor da complexidade de McCabe.

2.2 Algoritmos

Para esta etapa, decidimos mudar alguns algoritmos:

Estruturas aninhadas: Agora só será necessário verificar o que os ”filhos”retornam.

Variáveis não declaradas/inicializadas: Adicionando tipos de erros à lista de *erros* e com o erro obtido pelo método *checkID*, basta comparar se esse erro está na lista.

Nodos dos Grafos: São criados com um id aleatório para não haver duas instruções iguais com o mesmo nodo.

CFG: No momento da criação deste grafo, cada classe retorna um tuplo com 3 elementos. O primeiro é o id do nodo que o ”Pai”se deve ligar, o segundo é uma string referente ao grafo em si e o último é uma lista de ”endpoints”que se deverão ligar à próxima instrução.

SDG: Para este grafo cada classe retorna 4 valores: o primeiro é o id que se deve ligar ao ”Pai”, o segundo é o código referente a este grafo, o terceiro é a subtração entre arestas e nodos criados na instrução (e dos filhos) e o último é uma lista de nomes de funções cujo valor da Complexidade de McCabe depende destas.

Complexidade de McCabe: À medida que se vai adicionando os nodos e as arestas, vai-se contando. No fim, calcula-se com base nos valores das funções de que dependem e com o seu próprio valor.

Capítulo 3

Codificação e Testes

3.1 Testes realizados e Resultados

Mostram-se a seguir um dos testes feitos (ficheiro de *input*) e o respectivo resultado obtidos:

3.1.1 Teste

Figura 3.1: Resultado 1

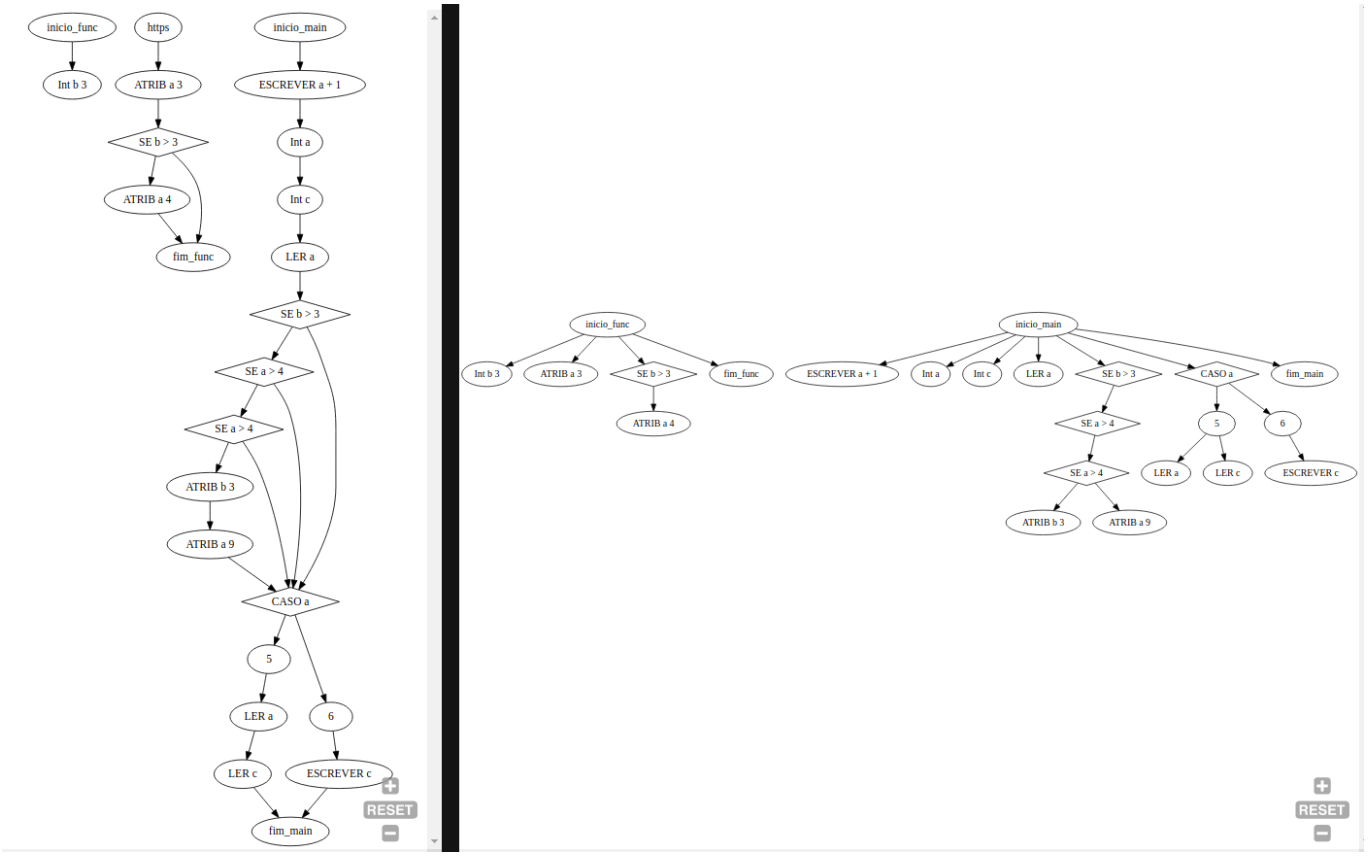

Figura 3.2: Grafos Correspondentes

Capítulo 4

Conclusão

4.1 Síntese do Documento

Ao longo deste trabalho foi desenvolvido um analisador de código capaz de verificar o comportamento através de grafos e extrair um conjunto de informações de uma LPI por nós definida utilizando as ferramentas da biblioteca Lark.Interpreter do Python.

4.2 Análise crítica dos resultados

O coletivo tem a noção de que melhorou bastante em relação à etapa anterior, não só porque o código ficou mais simples de perceber, como os algoritmos usados e a qualidade do output dado. Temos noção de que há sempre por onde melhorar, quer nas estruturas de dados, quer no embelezamento de código, mas achamos que este trabalho cumpre os requisitos para que seja classificado de um muito bom projeto.

4.3 Trabalho futuro

O grupo tem a opinião de que este projeto pode ser melhorado em algumas vertentes. Quanto à pagina HTML, as estatísticas poderiam estar numa barra lateral, de modo a não esconder uma parte do código, poderia haver um modo noturno e a escolha de cores talvez fosse outra. Já na gramática, poderia implementar outras possibilidades, como estruturas de dados ou importações de ficheiros.