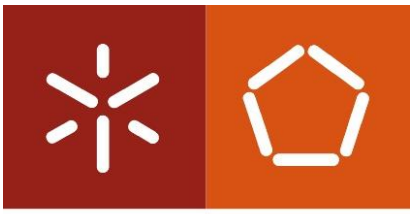

Universidade do Minho

Mestrado em Engenharia Informática

Engenharia Gramatical

Grafos na análise e interpretação de código fonte

Grupo 3

Duarte Parente (PG53791) Gonçalo Pereira (PG53834)
José Moreira (PG53963)

Ano Letivo 2023/2024

Índice

Capítulo 1

Introdução

O presente relatório visa apresentar o terceiro trabalho prático no âmbito da unidade curricular de **Engenharia Gramatical**. Esta etapa subsequente constitui a continuação do projeto submetido no segundo trabalho prático. Vale ressaltar que o objetivo do segundo trabalho prático passava por desenvolver um analisador de código fonte para uma linguagem de programação imperativa (LPI) desenvolvida pelo grupo, contando com alguns requisitos mínimos como permitir declarar variáveis atómicas e estruturadas (incluindo as estruturas: conjunto, lista, tuplo, dicionário), instruções condicionais e, pelo menos, três variantes de ciclos.

Neste terceiro trabalho prático, o objetivo proposto é enriquecer o analisador estático criado, estudando também o comportamento dos programas-fonte com base na construção dos vários **DAG** (**Directed Acyclic Graph**) que se usam para estudar o fluxo da execução (controlo) e dos dados (em função das dependências entre as variáveis). Em suma, deve ser escrito, 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 linguagem LPI desenvolvida pelo grupo e que gere, em formato *dot*, os respetivos grafos.

Este documento descreve todo o processo de criação de grafos de execução dos programas, contando também com exemplos de utilização e uma análise aos resultados obtidos, terminando com uma conclusão final.

Capítulo 2

Contextualização

De forma a relembrar melhor o trabalho desenvolvido pelo grupo no segundo trabalho prático, uma vez que este serviu como base para o desenvolvimento do presente trabalho, este capítulo servirá para fazer uma descrição um pouco mais detalhado da LPI e do que foi o interpretador que foi desenvolvido.

Relativamente à linguagem de programação imperativa desenhada pelo grupo, procurou-se a aplicação de uma sintaxe que permitisse conciliar a simplicidade de escrita e leitura, com os requisitos propostos para o desenvolvimento da mesma. Dessa forma, a linguagem C assumiu o papel de principal fonte de inspiração, com algumas noções baseadas em outras linguagens imperativas, especialmente o Python. Esta linguagem permite os tipos de dados `Int`, `Float`, `String` e `Boolean`, e as estruturas dicionários, listas, tuplos e `sets` (decalração, atribuição e acesso). Quanto a ciclos, os três permitidos são `for`, `while` e `do while`. Por último, permite também *scripted code* (ou seja, código fora de funçóes), e a declaração de funções (através da `keyword` "function"e o seu tipo de retorno).

No que toca ao interpretador, toda a sua estratégia de implementação encontra-se explicada detalhadamente no relatório anterior, porém vale relembrar que este conta com os dados recolhidos de cada função do programa a ser interpretado, dados acerca do *scripted code* e também algumas estatísticas globais. No que toca às estatísticas globais, é apresentado o número de ocorrências de variáveis por tipo, o cabeçalho de cada função (juntamente com o tipo de argumentos), o número ocorrências de cada tipo de instrução, o número de estruturas aninhadas e os número de `if` aninhados que podem ser substituídos por apenas um. Quanto às estatísticas do *scripted code* e das funções, são apresentados os erros, os *logs* das variáveis e o número de ocorrências de cada tipo de instruções. Toda esta informação pode ser também apresentada num ficheiro HTML, proporcionando ao utilizador uma apresentação mais estruturada e fácil de compreender.

Capítulo 3

Construção de Grafos

Tal como referido na introdução, o objetivo proposto é fazer com que a ferramenta permita também a construção de **CFG** (*Control Flow Graph*) para as seguintes instruções suportadas pela linguagem:

- Estruturas cíclicas (`for`, `while` e `do while`);
- Estrutura condicional `if-else`;
- Instruções de declaração, atribuição e input/output.

Para além destas instruções, o grupo conseguiu expandir ainda mais o conteúdo abrangido pelo grafo, fazendo com que todas os tipos de instruções permitidos no corpo da função sejam suportados (operações unárias, operações binárias, chamada de funções, etc.)

3.1 Estratégia de implementação

A estratégia adotada pelo grupo para a concretização do objetivo passou pela criação de três novas variáveis para a classe do interpretador: `graph`, `graph_acc` e `in_content`.

A primeira variável, `graph`, serve para guardar a string relativa ao grafo gerado pois, como sabemos, a biblioteca **graphviz** gera o grafo a partir de texto, sendo que o texto tem de ter determinadas características. A título de exemplo, a figura seguinte mostra como funciona esta conversão de texto para grafo.

```
digraph G {
  inicio -> "if x"
  "if x" -> "z=2"
  "z=2" -> "z=z+1"
  "if x" -> "z=z+1"
  "z=z+1" -> "fim"
  "if x" [shape=diamond];
}
```

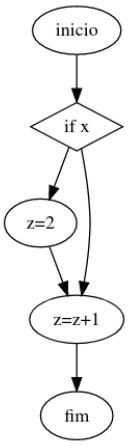

Figura 1: Exemplo - Grafo gerado a partir do texto apresentado

A segunda variável mencionada anteriormente, `graph_acc`, serve como um acumulador para cada instrução. Ou seja, supondo que o analisador está a interpretar a instrução `int a = 4`, no final da instrução esta variável vai ter o valor `graph_acc = "int a = 4"`. Por último, a variável `in_content` serve como uma variável de controlo para saber se é necessário fazer *reset* do acumulador. Por outras palavras, este acumulador passa por um *reset* no fim de cada instrução. Porém, há casos em que se quer que o acumulador continue a guardar informação. Por exemplo, um ciclo **for** é constituído por, entre outros, uma atribuição/declaração (p.e., for (**int i = 0**...) e, neste caso, não se pretende apagar o conteúdo do acumulador após essa mesma atribuição/declaração. É, portanto, nestes casos, que se usa a variável **in_content** que, deste modo, permite saber se se deve proceder ao *reset* do conteúdo do acumulador.

O grupo optou, depois, por guardar esse texto num ficheiro (na diretoria "data") e, com o ficheiro "dataToGraph", esse texto é convertido para uma imagem. Para isso, são usadas as bibliotecas **graphviz** e **PIL**.

Capítulo 4

Análise de Resultados

No presente capítulo, apresentam-se imagens com exemplos do output gerado pelo programa desenvolvido pelo grupo, para cada input. Note-se que, quando um programa tem mais do que uma função, o grafo transita diretamente de uma função para a outra, pois o importante é mostrar a execução do programa como um todo.

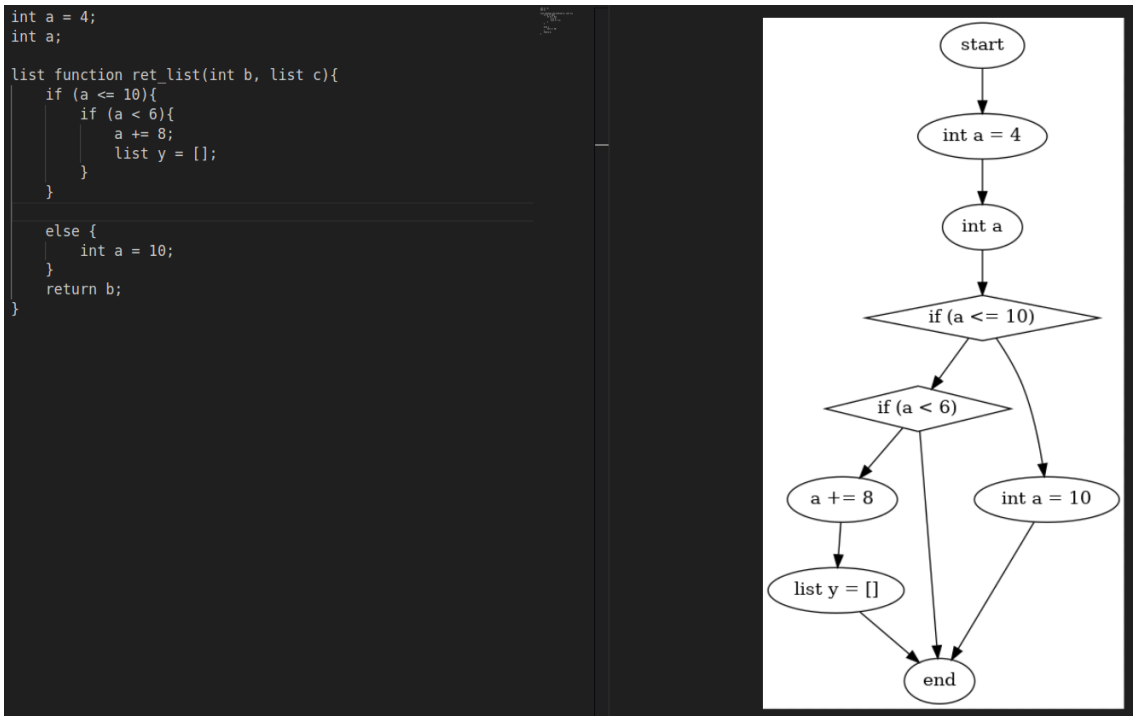

Figura 2: Exemplo 1 - Grafo gerado pelo programa desenvolvido

```
dict a = {'lionel': 'messi'};

while (a < 10){
    int b = 5;
    tuple f = (1, 2, 3);

    int j = b + f;
}

void function print(int a, dict b){
    dict b = {'yes': 'no'};

    if (j == b){
        print(a);
    }
}
```

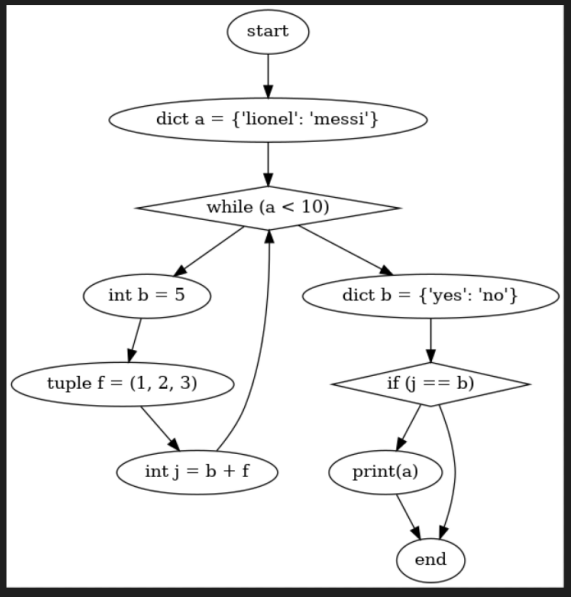

Figura 3: Exemplo 2 - Grafo gerado pelo programa desenvolvido

```
int a = 30;
a = 25;

int function try(int a, dict b){
    a += 35;
    for (int c = 10; c < 20; c = c + a;){
        if (c == 40){
            dict a = (1, 2, 3);
            print(a);
        }
        int y = 10;
    }
    string y = '2das';

    return y;
}
```

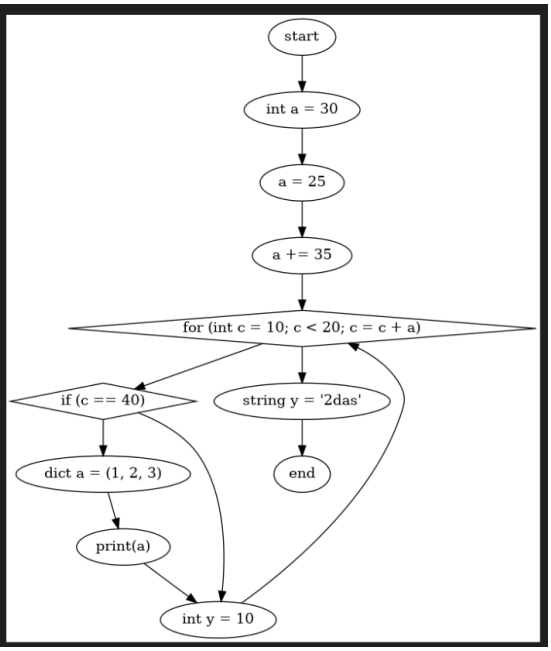

Figura 4: Exemplo 3 - Grafo gerado pelo programa desenvolvido

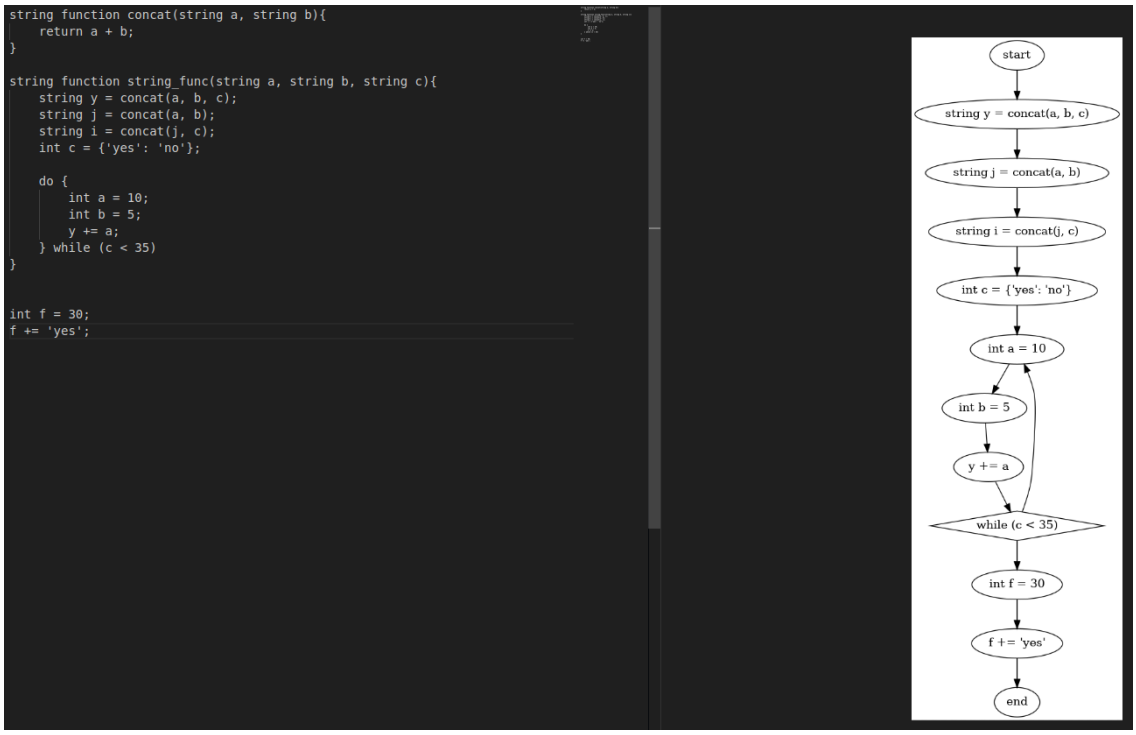

Figura 5: Exemplo 4 - Grafo gerado pelo programa desenvolvido

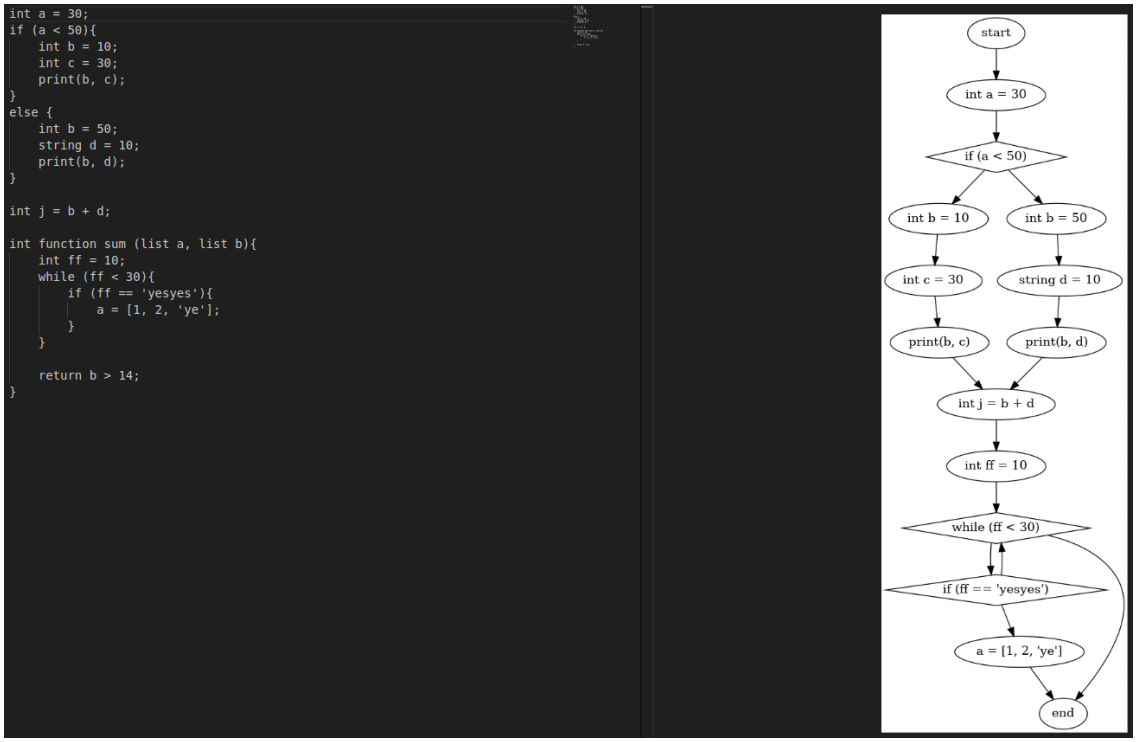

Figura 6: Exemplo 5 - Grafo gerado pelo programa desenvolvido

Capítulo 5

Conclusão

O objetivo principal deste terceiro e último trabalho prático foi enriquecer a ferramenta desenvolvida no segundo trabalho prático, de forma a implementar uma funcionalidade que, de certa forma, automatizasse a construção de grafos para estudar o fluxo de execução de um determinado programa.

Através dos exemplos mostrados acima, é possível demonstrar a capacidade desta nova funcionalidade do analisador, oferecendo ao utilizador uma forma bastante mais rápida e eficiente de analisar um programa, fornecendo várias vantagens que vão desde identificação de bugs a correção dos programas. Estas consequências culminam numa grande poupança de tempo, num desenvolvimento de um projeto.

Relativamente ao desenvolvimento futuro, as implementações relativamente à linguagem desenvolvida foram também enumeradas no relatório anterior. Partindo dessas alterações, o grupo pretende então adicioná-las à criação de grafos, criando assim grafos mais completos.

É, deste modo, que se encerra o atual relatório relativo ao terceiro trabalho prático desenvolvido, proposto pela unidade curricular de **Engenharia Gramatical**.