Engenharia Gramatical (1º ano MEI)

Trabalho Prático 3

Relatório de Desenvolvimento

Gonçalo Ferreira (pg50404)

Rui Braga (pg50743)

4 de junho de 2023

Resumo

O terceiro trabalho prático, de Engenharia Gramatical consistiu na geração de grafos de controlo de fluxo e de dependência de sistema a partir da linguagem de programação imperativa e do analisador de código já criados no segundo trabalho prático.

Esse relatório descreverá os métodos utilizadas, bem como, as pequenas modificações aplicadas ao trabalho anteriormente realizado.

No presente documento é descrito como foram gerados os grafos e, serão também apresentados os testes realizados de forma a demonstrar a capacidade do analisador.

Conteúdo

Capítulo 1

Introdução

Área: Processamento de Linguagens

Supervisor: Prf. Pedro Rangel Henriques e Tiago João Fernandes Baptista

Parte integrante da componente prática da UC de Engenharia Gramatical (integrada no perfil de Engenharia de Linguagens, no mestrado de Engenharia Informática), este relatório servirá de documentação do processo e resultados da produção do terceiro e último trabalho prático. Na sequência do segundo TP, este projeto pede agora do grupo, o aprimoramento do relatório de análise de código (desenvolvido na fase anterior), com a adição de grafos de controlo de fluxo e de dependências do sistema.

Para o efeito, o grupo terá de recorrer à notação dot e ao módulo python "graphviz", para através da informação recolhida pelo analisador de código, ser capaz de criar os ficheiros .dot e a vizualização dos mesmos.

Assim este documento começará por explorar a implementação dos "Control Flow Graphs", passando para os "System Dependency Graphs", demonstrando em ambos os resultados obtidos em alguns excertos de código criados para o efeito. Será também adicionada uma secção com algumas melhorias e conteúdos extra adicionados sob o 2º trabalho prático, que o grupo considerou uma mais-valia adicionar.

Capítulo 2

Grafos de Controlo de Fluxo

2.1 Implementação

Para a criação dos "Control Flow graphs", foi utilizada a estrutura de dados retornada pela análise de código, que converte o código, numa estrutura de fácil processamento e que descreve todas as instruções e declarações realizadas no código (estrutura esta que não sofreu alterações desde a última fase do projeto).
Assim, o grupo desenhou os seus CFGs atendendo as seguintes regras:

- Quando for encontrada uma declaração de variável ou utilização de uma função, deve ser criado um novo nodo, e todas as arestas anteriores deverão ligar-se a este. O anterior ao próximo nodo é estabelecido como o nodo da declaração.

- Quando for encontrado um ciclo (quer seja um *for loop*, *while loop* ou *do while*), depois de ser criado o nodo, e serem establecidas as ligações com os anteriores, o código contido dentro do loop deverá ser processado; o grafo gerado neste excerto, deverá ter inicio e fim no nodo de loop.

- Quando for encontrada um *"if"*, e após serem criados os nodos e as arestas anteriores, deverão ser processados, o conteúdo do "if"(ligado ao nodo do "if"e sem ligação de saída para já definida), e se existirem os *else if* e *else*(adicionando o nodo, e o grafo resultante do excerto). Deste processamento poderão sair vários ramos, que se deverão todos ligar ao mesmo próximo nodo (e são por isso guardados numa lista de identificadores de nodos).

- Quando for encontrado um "switch", o processo realizado é semelhante ao dos "ifs", com a diferença de que agora são percorridos todos os cases, e no máximo um default. Este processo também gerará múltiplos ramos, onde todos deverão estar ligados ao próximo nodo.

- Quando for encontrada uma definição de função, o contexto deverá mudar, sendo assim feita uma chamada recursiva à função geradora de grafos, desta vez, com um contexto diferente (para que seja gerado um novo grafo)

- No final do processamento de todas as instruções, os ramos resultantes, deverão ser ligados a um nodo final (se este tiver sido definido).

O grupo suportou-se no modúlo python graphviz para gerir a estrutura do grafo e renderizar o resultado final num formato de imagem (png).

2.2 Cálculo da Complexidade de McCabe

O cálculo da complexidade de McCabe (também conhecida como complexidade ciclomática é uma métrica de testes utilizada para medir a complexidade de um software, através da quantidade de caminhos independentes no código fonte.

A formula é $E - V + 2$, onde E é a quantidade de arestas existentes no grafo, e V a quantidade de vértices.

A implementação deste cálculo recorreu ao código de criação dos CFGs, que passou a necessitar de manter uma estrutura de dados que acumula a quantidade de nodos e arestas, para cada grafo dos diferentes contextos.

Capítulo 3

Grafos de Dependências do Sistema

3.1 Implementação

Para a criação dos grafos de dependências do sistema, foi novamente utilizada a estrutura de dados retornada pela analise de código, bem como o módulo Python *graphivz*.
Assim, o grupo desenhou os seus SDGs atendendo as seguintes regras:

- Cada grafo é iniciado com um nodo inicial, com a forma de um trapézio, chamado de "entry global"para o contexto global ou "entry {nome da função}"se o grafo descrever a definição de uma função. O grafo forma uma árvore, na qual este nodo inicial será a raíz e onde se ligam os restantes nodos na execução sequencial do programa.

- Quando for encontrado uma declaração de variável ou utilização de uma função, é criado um novo nodo, que se liga a um nodo pai, que pode ser o nodo inicial ou um nodo relacionado com estruturas de controlo de fluxo (loops, if ou switch).

- Quando for encontrado um ciclo (*for*, *while*, *do while*) é criado um nodo, que se torna nodo pai, dos nodos relativos ao código existente dentro do ciclo a ser processado.

- Quando for encontrado um *if*, são criados nodos "if"e "then"em que o nodo "if"é pai do nodo "then"e o nodo "then"é pai dos nodos relativos ao código que corresponde à situação em que a condição da estrutura de controlo que está a ser processada é verdadeira. Além disso, se existirem *elif* e *else*, também são criados nodos específicos para cada uma das situações, que se ligam ao nodo "if"e que são nodos pai dos nodos gerados pelo código relacionado com cada caso.

- Quando for encontrado um *switch*, é criado um nodo "switch"com a forma de um losango que se torna nodo pai de nodos "case"e um nodo "default"(caso exista) para cada situação descrita no mecanismo que esta parte do grafo está a descrever. Cada nodo "case"e "default"torna-se nodo pai dos nodos referentes às instruções que estão no seu contexto.

- Quando for encontrada uma definição de função, o contexto muda e é gerado um grafo novo, que considere o contexto da função e não o global, usando as regras acima definidas.

3.2 Identificação de Código Inalcançável

As zonas que o grupo identificou de código não alcançável são as instruções que são escritas depois de um *return* ter sido efetuado ou em mecanismos de controlo de fluxo em que certos casos nunca são explorados.

No 1º caso, basta ter uma *flag* que assinala que já foi feito *return*, de modo a que os seguintes nodos que são adicionados ao grafo, fiquem assinalados como inalcançáveis.

Para o 2º caso, são apenas assinaladas as situações em que se tem completa certeza que o código não é executado. Excluindo assim, condições que usem argumentos de variáveis, pois não é possível induzir o seu valor e condições dentro de ciclos, pois os valores das variáveis podem ser modificados durante a sua execução. De resto, caso seja possível calcular o valor da condição presente no mecanismo de controlo, e caso a condição seja falsa ou existir outra anterior que seja verdadeira (em estruturas com *elif* e *else*), esse código ficará sinalizado como inalcançável.

Os nodos relativos ao código inalcançável ficam assinalados nos grafos de dependência do sistema com arestas vermelhas. Como, por exemplo, num caso em que é óbvio que uma opção é sempre falsa (primeira condição $i == 2$) ou qual a opção escolhida ($i == 1$), tornando as outras inalcançáveis:

```
int i = 1;

if (i == 2){ write("OPÇÃO 1", "STDOUT"); }
elif (i == 1){ write("OPÇÃO 2", "STDOUT"); }
else{ write("OPÇÃO 3", "STDOUT"); }
```

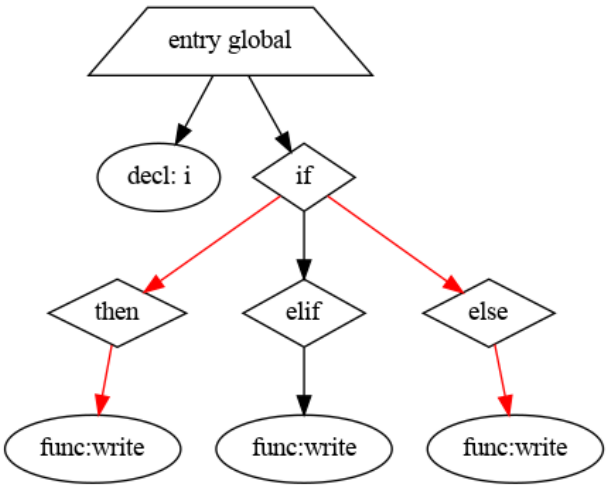

Figura 3.1: Exemplo da sinalização de código inalcançável por condições que nunca são verdade

O caso em que existem instruções depois de ter sido feito *return* pode ser demonstrado pelo exemplo seguinte:

```
def foo(){
        return 5;
        write("UNREACHABLE CODE", "STDOUT");
}
```

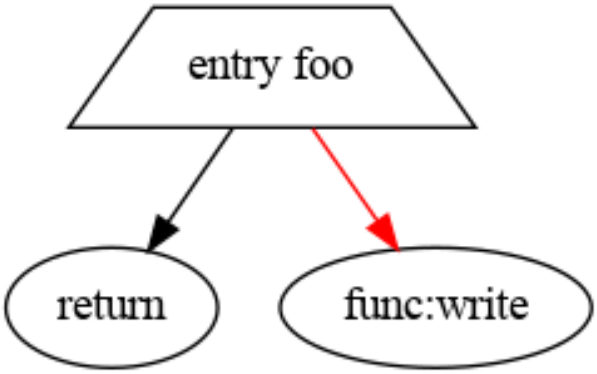

Figura 3.2: Exemplo da sinalização de código inalcançável depois de um return

Capítulo 4

Pequenas melhorias e Demonstração

4.1 Pequenas melhorias

- Foi melhorada a interface HTML do relatório produzido pelo analisador, com recurso à biblioteca de estilos CSS, Spectre. Foram também organizados os dados extraídos da análise, em dropdowns (com o auxílio de algum código javascript), para tornar o relatório mais intuitivo e navegável.
- Foram corrigidos alguns erros encontrados tanto no analisador como no gerador do relatório. A gramática não sofreu alterações.
- Foram tornados mais claros alguns dos warnings e erros, de forma a tornar mais percetível ao leitor, qual a verdadeira razão do erro.

4.2 Demonstração dos grafos

Para a demonstração dos grafos gerados o grupo passou o código abaixo para um ficheiro e executou a ferramenta com o mesmo:

```
def clock(){
    int time = 0;
    while(true){
        if (time % 2 == 0){
            write("Tic\n","STDOUT");
        }
        else{
            write("Tac\n","STDOUT");
        }
        time = time + 1;
    }
}
```

```
def stupid(){
    for i in [1 -> 10] {
      switch i
        case 0{
            write("-1","STDOUT");
        }
        case 1{
            write("0","STDOUT");
        }
        default {
            write(i,"STDOUT");
        }
    }
}
```

Que produziu os seguintes "Control Flow Graphs":

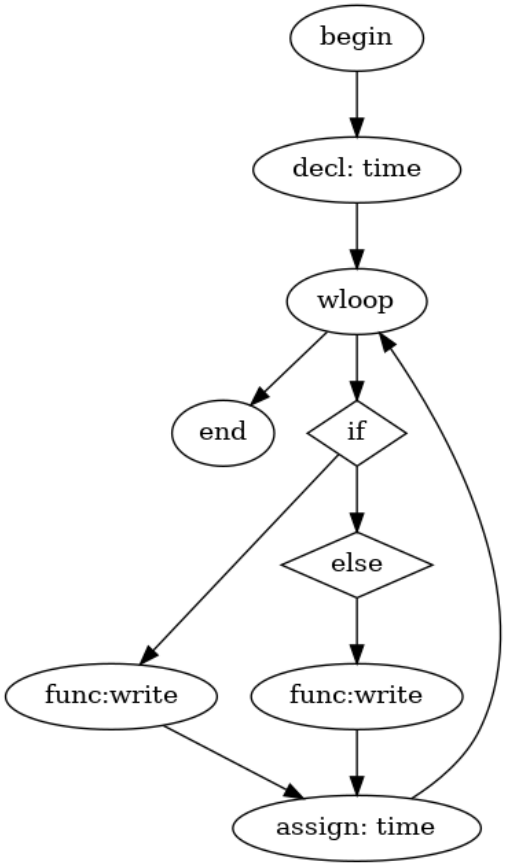

Figura 4.1: Grafo de controlo de fluxo gerado para a função clock

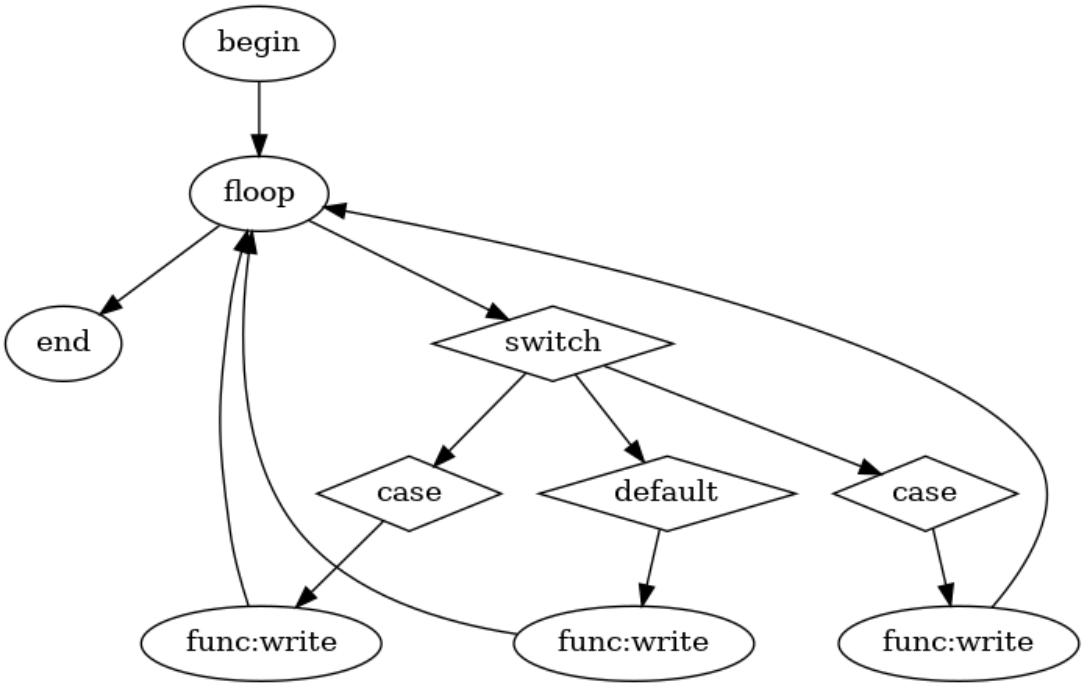

Figura 4.2: Grafo de controlo de fluxo gerado para a função stupid

E os seguintes "System Dependency Graphs":

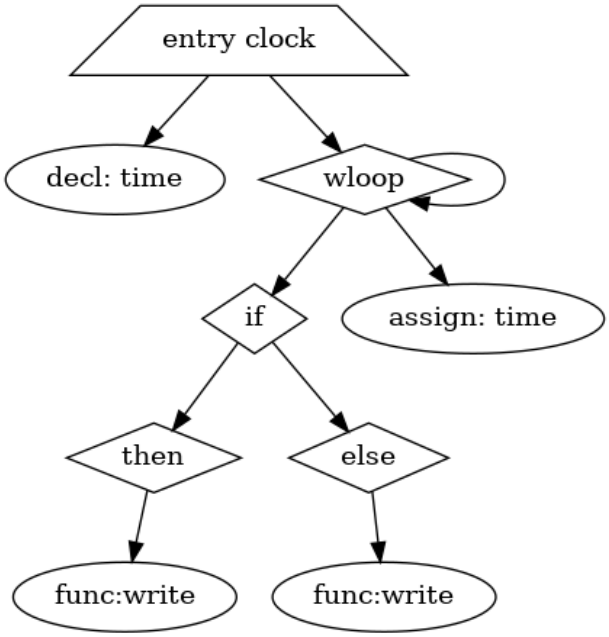

Figura 4.3: Grafo de dependências do sistema gerado a função clock

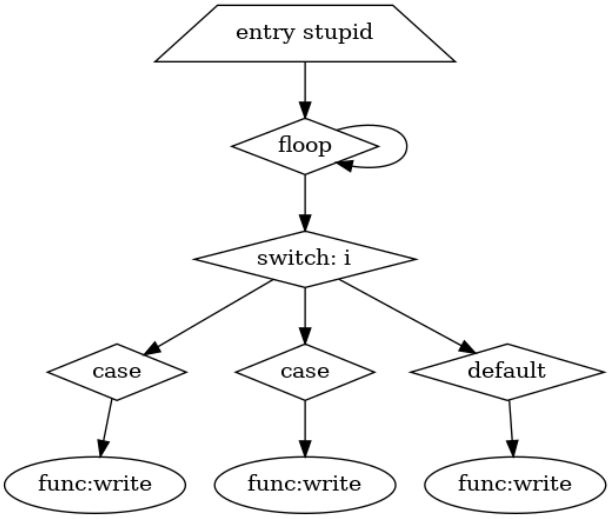

Figura 4.4: Grafo de dependências do sistema gerado a função stupid

Capítulo 5

Conclusão

Neste relatório abordamos as implementações do desenho de CFGs e SDGs para a linguagem criada, o cálculo da complexidade ciclomática de McCabe, zonas de código inalcançável, e algumas melhorias realizadas sob a fase anterior do projeto. Ainda assim o grupo gostaria de ter melhorado um pouco mais a interface da ferramenta, passando a mesma para um formato de webserver e de ter expandido as capacidades de anotação de código do analisador, com mais informações, mais detalhadas.

Apesar disto, o grupo faz uma avaliação positiva do trabalho produzido, acreditando que cumpre todos os requisitos impostos pelos docentes no enunciado, incluindo os objetivos opcionais (com a introdução de deteção de zonas inalcançável).

Apêndice A

Nova interface

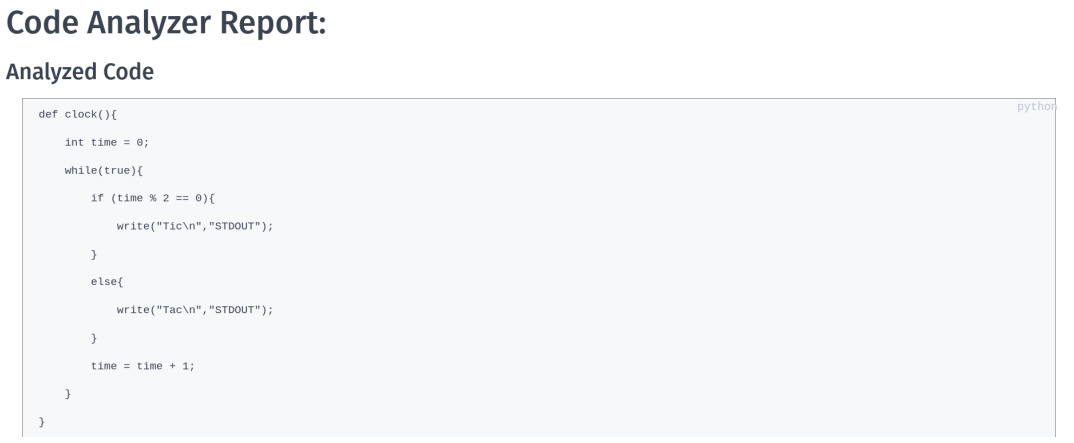

Figura A.1: Código anotado

Figura A.2: Resultados aninhados em Dropdowns

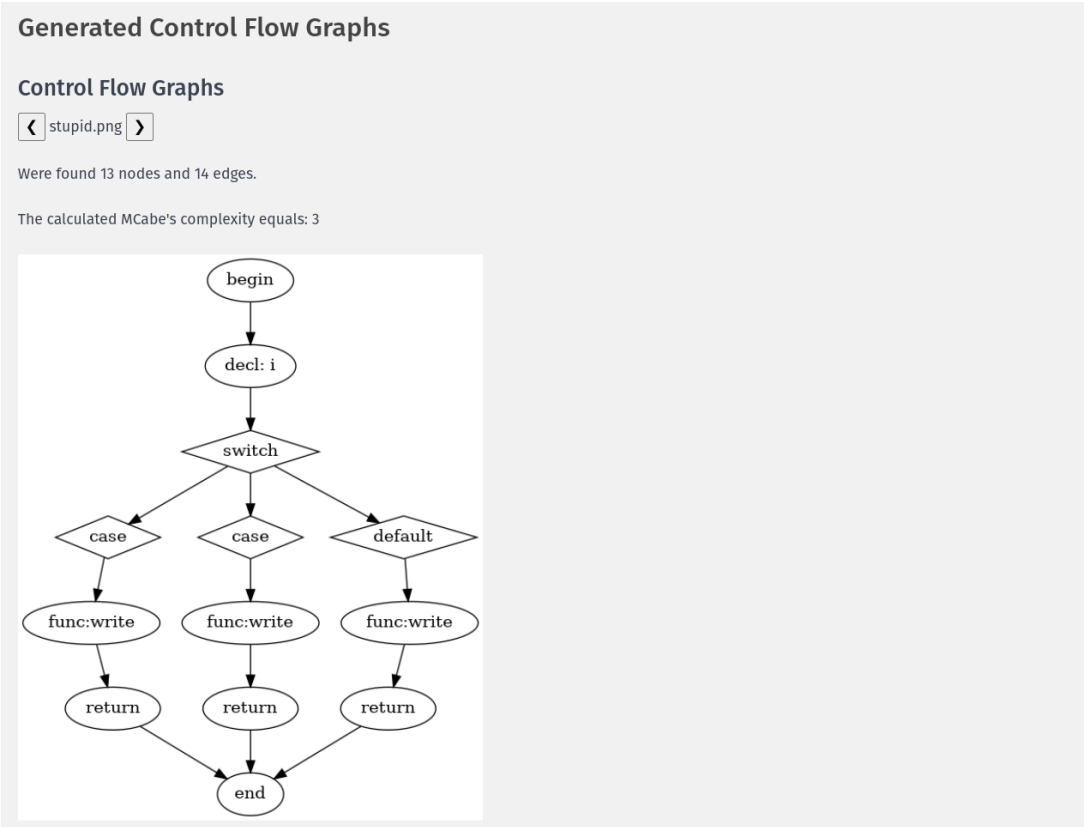

Figura A.3: Apresentação dos grafos