Cmb

Bruno Azevedo*and Miguel Costa†

Análise e Transformação de Software,
UCE30 Engenharia de Linguagens,
Mestrado em Engenharia Informatica,
Universidade do Minho

9 de Julho de 2012

Resumo

Este documento apresenta as resolução do Trabalho Prático de Análise e Transformação de Software em que se usa as técnicas de construção de ASTs e suas travessias tendo em conta atributos. O exercício está relacionado com a implementação da linguagem Cmb, dada nas aulas, e na criação de um Control Flow Graph (CFG), um Program Dependency Graph (PDG), System Dependency Graph (SDG) e ainda de um Directed Acyclic Graph (DAG). O resultado é um WebSite para mostrar os vários grafos criados.

*Email: azevedo.252@gmail.com
†Email: miguelpintodacosta@gmail.com

Conteúdo

1 Ambiente de Trabalho **3**

2 Descrição do problema **3**

3 Exemplo de um programa em Cmb **3**

4 Gramáticas **4**
4.1 Gramática concreta 4
4.1.1 AST 4
4.2 Tree Grammars 5
4.2.1 Control Flow Graph 5
4.2.2 Program Dependency Graph 11
4.2.3 System Dependency Graph 21
4.2.4 Single Static Assignment 25

5 WebSite **28**

6 Conclusões **33**

1 Ambiente de Trabalho

Foi necessário usar um Gerador de Compiladores para gerar o nosso próprio compilador, por isso usámos o AnTLR que é também usado nas aulas. Para facilitar o processo de debugging durante a resolução do problema dado, usámos a ferramenta AnTLRWorks, que tem uma interface bastante agradável e simpática para ajudar a resolver problemas desta natureza.

A linguagem de programação adoptada foi o JAVA. De forma a tornar a nossa solução mais legível e estruturada. Para mostrar a informação criamos um site com a linguagem PHP.

2 Descrição do problema

O pretendido para este exercício era gerar um pequeno WebSite, para navegar num Mapa de Conceitos, a partir de uma linguagem simples para descrever esses mesmos mapas.

A linguagem criada por nós tem de ser validada para depois ser apresentada a informação visualmente, terá ainda de haver uma forma de mostrar as ocorrências de cada conceito.

3 Exemplo de um programa em Cmb

Em todos os exemplos que vão ser apresentados neste relatório, o input introduzido é o que se segue:

Listing 1: Linguagem Cmb

```
1  void imprime(string nome)
2  {
3      string msg;
4      msg = "blabla";
5      print (nome);
6      print (msg);
7  }
8
9  int main(int x, int y)
10 {
11     int a;
12     int b;
13
14     a=2+1;
15
16     if(a==2) {
17         a = a + 1;
18         b = 3;
19     }
20     else
21         b = 5;
22
23     while (a<5)
24     {
25         a = a+5 * 3;
26     }
27
28     b = xtop(a+2*3,a);
29
30     print (a);
31
```

```
32      scan (a);
33  }
34
35  bool xtop (bool t, int i)
36  {
37      string s;
38
39      s = "luis";
40
41      imprime (s);
42
43      return s;
44  }
```

4 Gramáticas

Neste capítulo, iremos abordar a gramática concreta fornecida que define uma linguagem Cmb apresentada nas aulas de Engenharia Gramatical e explicar o objetivo da mesma. A transformação numa AST também foi fornecida e por isso abordaremos apenas o processo de criação associado. Por fim, mostraremos a implementação da cada um dos módulos (Tree Grammars) pedidos para este trabalho.

4.1 Gramática concreta

A linguagem Cmb é uma simplificação da linguagem C. Simplificada no sentido em que apenas reúne algumas das características presentes no C, nomeadamente, funções e os seus argumentos, declaração de variáveis, atribuições, operações de I/O (scan e print), expressões if e ciclos while, invocação de funções, retorno de variáveis numa função (return) e a utilização de expressões que utilizam operadores aritméticos, à exceção dos operadores de incremento (++) e decremento (–), operadores de comparação e operadores lógicos.

Após criada a gramática, o próximo passo seria validar o texto de input. Mas como fazê-lo? Existem várias abordagens, uma delas e a que iremos utilizar neste trabalho é a geração de uma representação intermédia para que a partir dela se possam criar os grafos pedidos. Na secção seguinte, iremos falar desta abordagem.

4.1.1 AST

Uma Representação Intermédia (RI) é uma versão independente de qualquer linguagem ou máquina do código original. A utilização de uma RI traz algumas vantagens tais como o aumento do nível de abstração e uma separação mais limpa entre o produto inicial e o final.

Existem várias representações intermédias e a que iremos utilizar é a AST (Abstract Syntax Tree) que é uma representação em árvore da estrutura sintática abstrata do código fonte. A sintaxe é abstrata no sentido em que não representa cada detalhe que aparece na sintaxe real, ou seja, elementos como parênteses de agrupamento estão implícitos na estrutura da árvore e uma construção sintática como uma condição if e os seus blocos then e else pode ser representada através de um único nodo e dois ramos, e símbolos intermédios e palavras reservadas são tipicamente eliminados. Basicamente, mantém-se uma estrutura suficiente para realizar processos semânticos e geração de código.

O próximo passo será, então, criar a AST e para isso é necessário criar regras de reescrita sobre a gramática concreta, um mecanismo que o ANTLR oferece. Enquanto que uma gramática de parsing especifica como reconhecer input, as regras de reescrita são gramáticas geradoras, ou seja, especificam como gerar output. Estas regras de reescritas, já nos são fornecidas juntamente com a gramática concreta e a AST resultante terá, por cada, elemento que agrupa outros elementos, um token imaginário, ou seja, referências a tokens que não se encontram na produção original, elementos tais como ';', ou parênteses serão eliminados e elementos com o mesmo nome numa produção são agrupados numa única lista.

A gramática final com as regras de reescrita pode ser consultada em anexo.

4.2 Tree Grammars

O próximo passo consiste na construção de um parser da AST gerada, que permitirá atravessá-la (tree walker) e manipulá-la, transformando-a gradualmente em diversas fases de tradução até que se obtenha uma forma final que satisfaça as nossas necessidades. Este parser será construído utilizando um mecanismo fornecido pelo ANTLR, uma Tree Grammar (TG). As ações numa TG possuem um contexto muito nítido e conseguem aceder a informação passada das regras invocadas.

A utilização de TGs, para além da utilização referida acima, também nos fornece algumas vantagens:

- uma especificação formal, concisa e independente de um sistema da estrutura da AST;
- as ações têm um contexto implícito graças à sua localização na gramática;
- os dados podem ser passados entre as ações de forma livre utilizando parâmetros (atributos), valores de retorno e variáveis locais.

O problema apresentado, exige a construção de 4 TG (módulos), por isso, dividiremos esta secção em 4 subsecções.

A primeira referente ao Control Flow Graph (CFG), o segundo ao Program Dependency Graph (PDG), o terceiro ao System Dependency Graph (SDG) e o último referente ao Single Static Assignment (SSA).

4.2.1 Control Flow Graph

Um Control Flow Graph (CFG) é um grafo direcionado com a adição de um único nodo de entrada START e um único nodo de saída STOP, tal que cada nodo no grafo tem no máximo dois sucessores. Assume-se que nodos com dois sucessores têm atributos "T"(Verdadeiro) e "F"(Falso) associados aos ramos que saem, assume-se também que, para cada nodo N no grafo, existe um caminho de START a N e um caminho de N a STOP.

Este tipo de grafos pode servir para verificar se os programas que representam estão corretos, verificando se determinados nodos são atingíveis ou não e a partir de aí otimizar o código, por exemplo, eliminando *dead code*, código que que é executado mas cujo resultado nunca é utilizado em mais nenhuma computação.

Numa visão geral , a estratégia utilizada para gerar este grafo consiste, na passagem de uma instância de uma classe Java Grafo (esta será explicada de seguida) por todas as produções da TG, assim como uma lista (`TreeSet<Integer>` em Java) com um ou mais números de instrução correspondentes à última(s) statement(s) executada(s) com o objetivo de conectar no grafo instruções consequentes e também, uma string label que indica se a instrução que está a ser calculada é a primeira instrução de um bloco then ou else. Todo o código externo à TG está escrito na linguagem de programação Java.

A classe Grafo é uma classe Java que armazena os nodos e os caminhos do CFG encontrados ao longo do processo de parsing da TG para cada função existente no programa input. Cada nodo é um mapeamento (TreeMap) entre o número de instrução (sequencial) e a instrução, que é representada por uma classe Instrucao, de momento, apenas importa referir que a classe Instrucao armazena a instrução em si. Cada caminho é um mapeamento entre o número da instrução origem e uma lista (`TreeSet<ParNrInstrucaoLabel>`), a classe `ParNrInstrucaoLabel` é constituída pelo número da instrução destino e a label referida acima. Um grafo é gerado para cada função existente no programa de input.

A lista com números de instrução é atualizada com o número da última instrução, no caso de uma atribuição, read, write, invocação ou return, ou das últimas instruções no caso de um if, while ou vários dos mesmos aninhados. Na instrução seguinte, a lista é utilizada para conectar os números contidas nela ao número da nova instrução visitada. A lista é inicializada com o número do nodo START (0) e após a última instrução de uma função ser executada, o conteúdo da lista é conectado ao nodo EXIT que é gerado na altura.

Finalmente, a string label é inicializada com uma string vazia e apenas toma o valor "T"quando o bloco then de um statement if ou quando o bloco de instruções de um while é visitado. E toma o valor "F"quando

o bloco else de um statement if é visitado ou o bloco de instruções de um while termina. Nos restantes casos, depois de um statement ser visitado esta volta a tomar o valor de string vazia.

No final da execução da TG um TreeMap contendo um mapeamento entre o identificador de uma função e o respetivo grafo CFG é retornado à Interface Java (esta será explicada posteriormente).

O processo de codificação das ações revelou-se, apesar de não ser idêntico em todas as produções com características semelhantes, sistemático. Após codificada a ação para uma produção, as restantes precisaram apenas de pequenas alterações. Tentaremos mostrar, com alguns excertos da TG, as ações criadas para obter o resultado final.

Começamos pela produção inicial, cujo símbolo não terminal representa o axioma da nossa linguagem, a partir da qual todo o processo de derivação se inicia. O TreeMap que vai armazenar os CFG gerados é instanciado nesta produção e para cada função existente no programa input, um novo grafo é instanciado e passado como atributo ao símbolo não terminal funcao, e o grafo resultante do processamento de uma função é adicionado ao TreeMap que será retornado no final. A produção é apresentada de seguida:

Listing 2: Produção programa

```
1     programa returns [TreeMap<String, Grafo> grafos_out]
2     @init {
3         TreeMap<String, Grafo> grafos = new TreeMap<String, Grafo>();
4     }
5         :   ^(PROGRAMA (funcao[new Grafo()]
6         {
7             grafos.put($funcao.func_id, $funcao.g_out);
8         }
9         )+
10        {
11            $programa.grafos_out = grafos;
12        }
13        )
14        ;
```

A ação criada para tratar cada função, mais especificamente o símbolo não terminal (SNT) corpo_funcao. Nesta produção, apenas na primeira visita, o TreeSet que guarda os números das últimas instruções executadas é instanciado, assim como o nodo inicial START é adicionado ao grafo e o número criado para este (0) é adicionado-o ao TreeSet. Num CFG as declarações de uma função são ignoradas, por isso apenas nos interessa a parte dos statements. Este símbolo não terminal recebe como atributos de entrada (herdados) para além dos três atributos já mencionados, uma string contexto que toma valores entre "CORPO_FUNCAO"e "BLOCO", o seu significado será explicado posteriormente. A produção é exibida de seguida:

Listing 3: Produção corpo funcao

```
1 corpo_funcao [Grafo g_in, String id_funcao] returns [Grafo g_out]
2     @init{
3         Grafo g = $corpo_funcao.g_in;
4         TreeSet<Integer> nrs = new TreeSet<Integer>();
5         // 0 <=> Nodo START. E passado como parametro para que o nodo START se ligue
              Ã  primeira instrucao
6         int nr = g.putNodo(0, new Instrucao("START", null, null));
7         nrs.add(nr);
8     }
9         : ^(CORPO declaracoes statements[$corpo_funcao.g_in, "CORPO_FUNCAO", nrs, ""
              ])
10        {
11            $corpo_funcao.g_out = $statements.g_out;
12        }
13        ;
```

Apenas mostramos a produção para o SNT statements para explicar a utilizada do atributo herdado contexto. Após todas as instruções de uma função terem sido executadas e devidamente introduzidas no grafo assim como os caminhos entre eles, falta conectar os nodos das últimas instruções ao nodo EXIT. Como esta

produção é visitada, não só a partir da produção corpo_funcao, mas também da produção bloco, que representa o conjunto de instruções presentes num bloco then ou else de um if ou as instruções de um bloco while, é necessário distinguir o contexto em que a produção é visitada, visto que só faz sentido ligar os último nodos ao nodo EXIT se realmente se tratar do final do corpo da função e não o final do bloco de um ciclo while.

Listing 4: Produção statements

```
statements [Grafo g_in, String contexto, TreeSet<Integer> nrs_ultima_instrucao_in,
    String label_in] returns [Grafo g_out, TreeSet<Integer> nrs_ultima_instrucao_out,
     String label_out]
    @init {
        Grafo g = $statements.g_in;
        TreeSet<Integer> nrs_ultima_instrucao = $statements.nrs_ultima_instrucao_in;
    }
        :    ^(STATEMENTS (statement[g, nrs_ultima_instrucao, $statements.label_in]
        {
            g = $statement.g_out;
            nrs_ultima_instrucao = $statement.nrs_ultima_instrucao_out;
            $statements.label_in = $statement.label_out;
        }
        )+
        {
            // Apos todos os statements do corpo de uma funcao tiverem sido
                executados, os ultimos nodos sao ligados ao nodo EXIT
            if ($statements.contexto.equals("CORPO_FUNCAO")){
                // cria nodo EXIT
                int nodo_exit = g.putNodo(new Instrucao("EXIT", null, null));
                // verifica se existem instrucoes anteriormente executadas e conecta
                    essas instrucoes ao nodo EXIT
                g.checkAndPutCaminho(nrs_ultima_instrucao, new ParNrInstrucaoLabel(
                    nodo_exit, $statement.label_out));
            }

            $statements.g_out = g;
            $statements.nrs_ultima_instrucao_out = nrs_ultima_instrucao;
            $statements.label_out = $statement.label_out;
        }
        )
        ;
```

Para o SNT statement, apenas ilustraremos a ação criada para um statement em específico visto que os restantes são semelhantes.

Vamos apresentar o caso em que o lado direito da produção contém o SNT atribuicao. Neste caso, os atributos passados a atribuicao são o grafo e a label e após esta ser executada um ou mais caminhos são adicionados ao grafo, isto é, a lista dos números das últimas instruções anteriores à atribuição são conectados ao número de instrução gerado para a atribuição e que é devolvido pelo SNT atribuicao através de um atributo sintetizado juntamente com a label passada como atributo herdado ao statement.

Listing 5: Produção statement

```
statement [Grafo g_in, TreeSet<Integer> nrs_ultima_instrucao_in, String label_in]
    returns [Grafo g_out, TreeSet<Integer> nrs_ultima_instrucao_out, String label_out
    ]
    @init {
        Grafo g = $statement.g_in;
    }
        :   atribuicao[g, $statement.label_in]
            {
                g = $atribuicao.g_out;

                // verifica se existem instrucoes anteriormente executadas e conecta
                    essas instrucoes a nova instrucao
```

```
10                  g.checkAndPutCaminho($statement.nrs_ultima_instrucao_in, new
                        ParNrInstrucaoLabel($atribuicao.nrs_ultima_instrucao_out.first(),
                         $statement.label_in));
11
12                  $statement.g_out = g;
13                  $statement.nrs_ultima_instrucao_out = $atribuicao.
                        nrs_ultima_instrucao_out;
14                  $statement.label_out = $atribuicao.label_out;
15              }
16          | ...
17          ;
```

Este comportamento só modifica para as produções cujo lado direito da produção são o SNT ifs e whiles, sendo que a única diferença é o facto de a adição de novos caminhos ao grafo não ser feita após a execução da instrução, mas sim nas produções correspondentes. Devido a esse facto a essas produções também lhes é passado como atributo herdado a lista dos números das últimas instruções.

Aproveitando o exemplo dado acima, vamos ilustrar a produção do SNT atribuição e ação criada para o mesmo. Esta ação consiste na inserção de um novo nodo no grafo, que como já foi dito, apenas consiste na inserção da instrução em si, assim como a atualização do TreeSet que armazena a última instrução executada, neste caso, este número é retornado pelo método de inserção de um novo nodo no grafo. O TreeSet e o grafo são retornados como atributos sintetizados.

Listing 6: Produção atrbuicao

```
1  atribuicao [Grafo g_in, String label_in] returns [Grafo g_out, TreeSet<Integer>
       nrs_ultima_instrucao_out, String label_out]
2      @init {
3          Grafo g = $atribuicao.g_in;
4      }
5          :     ^('=' ID expr)
6          {
7              TreeSet<Integer> nrs = new TreeSet<Integer>();
8              // cria nodo no grafo e guarda o nr da instrucao
9              nrs.add(g.putNodo(new Instrucao($ID.text + " = " + $expr.instrucao, null,
                   null)));
10
11             $atribuicao.nrs_ultima_instrucao_out = nrs;
12             $atribuicao.g_out = g;
13             $atribuicao.label_out = "";
14         }
15         ;
```

Da mesma forma que os casos do ifs e whiles diferiam dos outros, também a este nível diferem. As produções para ambos são um pouco mais elaboradas que as restantes.

Para o caso do ifs é preciso ter em atenção que os últimos nodos executados ligam apenas à expressão de condição do if, e esta, por sua vez, liga à primeira instrução do bloco then e, caso exista, à primeira instrução do bloco else, se não existir então é necessário conectá-la a próxima instrução a ser executada. Posto isto, para facilitar as explicações seguintes, mostraremos de seguida a produção no seu formato original na AST para servir de guia.

Listing 7: Produção ifs

```
1  ifs
2          : ^(IF expr a=bloco (b=bloco)?)
3          ;
```

O primeiro passo é a criação do nodo correspondente à expressão do if no grafo, assim como, dos caminhos entre as últimas instruções a ser executadas e a instrução correspondente à expressão, juntamente com a label passada como atributo herdado. Adicionalmente, o número da nova instrução é atribuído a duas variáveis auxiliares cuja utilidade está explicada nos comentários do excerto de código seguinte.

Listing 8: Produção ifs

```
ifs [Grafo g_in, TreeSet<Integer> nrs_ultima_instrucao_in, String label_in] returns [
    Grafo g_out, TreeSet<Integer> nrs_ultima_instrucao_out, String label_out]
    @init {
        Grafo g = $ifs.g_in;
        int nr_ult_inst_exp = -1;
        TreeSet<Integer> nrs_exp = new TreeSet<Integer>();
        TreeSet<Integer> nrs_out = new TreeSet<Integer>();
    }
        :   ^(IF expr
                {
                    // cria nodo no grafo e guarda o nr da instrucao
                    nr_ult_inst_exp = g.putNodo(new Instrucao($IF.text + "(" + $expr.
                        instrucao + ")", null, null));

                    // verifica se existem instrucoes anteriormente executadas e
                        conecta essas instrucoes a nova instrucao (expressao)
                    g.checkAndPutCaminho($ifs.nrs_ultima_instrucao_in, new
                        ParNrInstrucaoLabel(nr_ult_inst_exp, $ifs.label_in));

                    // variavel que sera passada aos blocos para indicar o nodo que
                        sera ligado a primeira instrucao de cada bloco
                    nrs_exp.add(nr_ult_inst_exp);

                    // adiciona provisoriamente o nr da expressao. caso exista o
                        bloco else, este e removido ja que a instrucao seguinte,
                        ligar-se-a a Ãºltima instrucao dos blocos then e else
                    // caso contrario, ligar-se-a a expressÃ■o e ao bloco then.
                    nrs_out.add(nr_ult_inst_exp);
                }
```

Ao bloco de instruções then são passados como atributos herdados o grafo, o TreeSet com o número da instrução correspondente à expressão do if, ou seja, a última instrução executada e a string ”T”, que corresponde à label que indica que a próxima instrução será conectada à expressão com a label ”T”. Após a sua execução o grafo e o TreeSet das últimas instruções são atualizados com os atributos sintetizados pelo bloco.

Listing 9: Produção bloco

```
        a=bloco[g, nrs_exp, "T"]
        {
            g = $a.g_out;
            // adiciona os nrs das utlimas instrucoes deste bloco
            nrs_out.addAll($a.nrs_ultima_instrucao_out);
        }
```

O bloco de instruções else é semelhante ao bloco anterior, apenas difere no atributo herdado label que passa a ser ”F”e, tal como, foi dito acima, o TreeSet nrs_out tinha guardado temporariamente o número da instrução da expressão do if, se o bloco else for visitado significa que este existe, logo não faz sentido que a instrução da expressão seja conectada à próxima instrução.

Listing 10: Produção bloco

```
(b=bloco[g, nrs_exp, "F"]
            {
                g = $b.g_out;
                // remove o nr da expressao adicionado provisoriamente
                nrs_out.remove(nr_ult_inst_exp);
                // adiciona os nrs das utlimas instrucoes deste bloco
                nrs_out.addAll($b.nrs_ultima_instrucao_out);
            } )?
        )
        {
```

```
            $ifs.nrs_ultima_instrucao_out = nrs_out;
            $ifs.g_out = g;
            $ifs.label_out = "";

        }
    ;
```

Para o caso do whiles, temos que, à semelhança do ifs, os últimos nodos executados ligam apenas à expressão de condição do while, e esta liga à primeira instrução do bloco e à próxima instrução a ser executada após o bloco. Tal como fizemos anteriormente, para facilitar as explicações seguintes, mostraremos a produção no seu formato original na AST para servir de guia.

Listing 11: Produção whiles

```
whiles
        : ^(WHILE expr bloco)
        ;
```

O primeiro passo é semelhante ao do if, ou seja, criamos o nodo correspondente à expressão do while no grafo, assim como, dos caminhos entre as últimas instruções a ser executadas e a instrução correspondente à expressão. E o número da nova instrução é guardado numa variável que será depois passada ao bloco.

Listing 12: Produção whiles

```
whiles [Grafo g_in, TreeSet<Integer> nrs_ultima_instrucao_in, String label_in]
    returns [Grafo g_out, TreeSet<Integer> nrs_ultima_instrucao_out, String label_out
    ]
    @init {
        Grafo g = $whiles.g_in;
        int nr_ult_inst_exp = -1;
        TreeSet<Integer> nrs_exp = new TreeSet<Integer>();
    }
        :    ^(WHILE expr
            {
                // cria nodo no grafo e guarda o nr da instrucao
                nr_ult_inst_exp = g.putNodo(new Instrucao($WHILE.text + "(" + $expr.
                    instrucao + ")", null, null));

                // verifica se existem instrucoes anteriormente executadas e conecta
                    essas instrucoes a nova instrucao (expressao)
                g.checkAndPutCaminho($whiles.nrs_ultima_instrucao_in, new
                    ParNrInstrucaoLabel(nr_ult_inst_exp, $whiles.label_in));

                // variavel que sera passada ao bloco para indicar o nodo que sera
                    ligado a primeira instrucao do bloco
                nrs_exp.add(nr_ult_inst_exp);
            }
```

Ao bloco de instruções são passados como atributos herdados o grafo, o TreeSet com o número da instrução correspondente à expressão do while e a string ”T”. Após a sua execução o grafo é atualizado com o grafo passado como atributo sintetizado pelo bloco.

Finalmente, novos caminhos entre as últimas instruções do bloco e a expressão são inseridos no grafo. O TreeSet com o número da instrução correspondente à expressão de condição, o grafo e label com o valor ”F”para que a próxima instrução seja conectada com essa label são retornadas como atributos sintetizados.

Desta forma, apresentámos a implementação da solução encontrada para construir um ou mais CFG a partir de um programa Cmb. A Tree Grammar completa (CmbTGCFG.g) pode ser consultada em anexo.

Exemplo de um grafo CFG gerado:

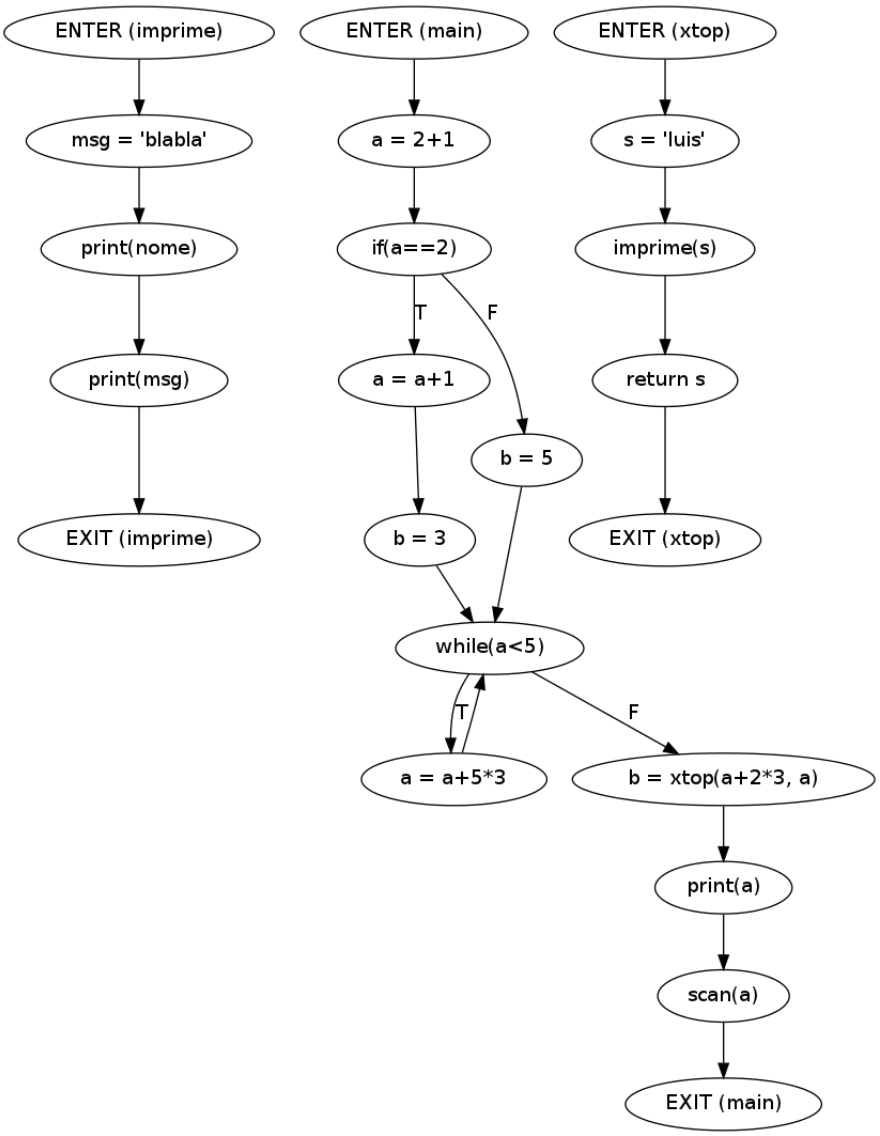

Figura 1: Grafo CFG

4.2.2 Program Dependency Graph

Um Program Dependency Graph (PDG) é uma representação de um programa que contém os mesmos nodos que o CFG e dois tipos de arestas: arestas de controlo de dependência e arestas de controlo de dados. Portanto, é um grafo direcionado em que os vértices correspondem a statements e predicados de controlo, e as arestas correspondem a dados e a controlos de dependência.

Dado que as dependências neste grafo conectam partes do programa computacionalmente relevantes, muitas transformações para melhoramento de código requerem menos tempo de execução comparativamente a outras representações de programas. Uma simples travessia sobre estas dependências é suficiente para realizar bastantes otimizações. Dado que ambos os tipos de de dependências estão presentes numa única forma, transformações como vetorização[1] conseguem lidar com as dependências de controlo e dados de uma forma uniforme. Transformações tais como *code motion*[2], que requerem a interação dos dois tipos de dependência, também conseguem ser controladas por este grafo. Para além disso, a natureza hierárquica do PDG permite que grandes secções

[1]Vetorização é o processo de converter um programa de uma implementação escalar que processa um único par de operandos de cada vez, para uma implementação vetorial que processa uma operação em múltiplos pares de operandos.

[2]Code motion consiste na movimentação automática, por parte do compilador, de statements ou expressões presentes num ciclo para fora do mesmo sem afetar a semântica do programa.

do programa sejam sumariadas. É por isso, a base de muitos algoritmos eficientes para transformações de reordenação.

Vamos dividir a explicação da nossa estratégia em duas partes, correspondentes à implementação das dependências de controlo e dependências de dados.

Dependências de controlo

As dependências de controlo diferem um pouco das do CFG, no sentido em que o fluxo das instruções é representado da esquerda para a direita, sendo o nodo pai o nodo que representa a entrada na função e os restantes nodos, correspondentes às instruções no contexto desta função, são conectados a este nodo inicial. Quando um ciclo while ou uma estrutura if são encontrados, é gerada uma sub-árvore, sendo a raiz desta sub-árvore a expressão de controlo da estrutura. Por isso, a estratégia passa também por fazer passar como atributo uma instância de uma classe GrafoPDG (herdado: GrafoPDG g_in; sintetizado: GrafoPDG g_out) por grande parte da AST, uma lista (herdado: TreeSet<Integer> nrs_ultima_instrucao_in; sintetizado: TreeSet<Integer> nrs_ultima_instrucao_out) com o número da instrução inicial, ou quando o contexto mudar (quando um ciclo while ou uma estrutura de controlo if for encontrado) o número da instrução do predicado de controlo dessa estrutura, com o objetivo de conectar as instruções de modo a representar o fluxo de dados pretendido e, finalmente, uma string label (herdado: String label_in; sintetizado: String label_out), "T"se a instrução pertence a um bloco then ou ao bloco de instruções de um while e "F"se pertence a um bloco else. Esta label estará depois associada a uma aresta entre um predicado de controlo e uma instrução.

A classe GrafoPDG é uma extensão à classe Grafo, herdando as variáveis e métodos da classe Grafo, adicionalmente possui uma variável que representa as dependências de dados numa função, ou seja, é um TreeMap que conecta um número de instrução a outros, cujas variáveis referenciadas nessas instruções são dependentes da variável definida na primeira. Mais tarde, quando as dependências de dados forem abordadas, será explicado com mais detalhe esta implementação. Também neste módulo, um grafo é gerado para cada função existente no programa de input.

A lista com o número da última instrução é atualizado sempre que o contexto muda, ou seja, quando uma estrutura while ou if são executadas. Em cada instrução, a lista é utilizada para conectar o número contido nela ao número da nova instrução executada. A lista é inicializada com o número do nodo inicial (0).

Finalmente, a string label é inicializada com uma string vazia e apenas toma o valor "T"quando o bloco then de um statement if ou quando o bloco de instruções de um while é visitado. E toma o valor "F"quando o bloco else de um statement if é visitado. Nos restantes casos, depois de um statement ser visitado esta volta a tomar o valor de string vazia. No final da execução da TG um TreeMap contendo um mapeamento entre o identificador de uma função e o respetivo grafo PDG é retornado à Interface Java (esta será explicada posteriormente).

A implementação das ações associadas à deteção de dependências de controlo, por ser semelhante à do CFG, não será ilustrada. A Tree Grammar pode ser consultada em anexo.

Dependências de dados

Uma dependência de dados existe entre duas instruções sempre que uma variável que é referenciada numa delas possa ter um valor incorreto se a ordem das instruções for trocada. A estratégia encontrada para detetar estas dependências pode ser dividida em duas abordagens distintas:

- Para cada instrução visitada, uma procura de dependências de dados é iniciada a partir dessa instrução comparando com outras instruções antecedentes;
- Sempre que um ciclo while for visitado, a procura de dependências para as instruções pertencentes ao seu bloco são adiadas até que o final do bloco seja atingido, altura em que uma procura a estas instruções é feita com um algoritmo diferente ao anterior, visto tratar-se de um ciclo e a ordem das instruções ser afetada;

Para que as abordagens referidas acima fossem possíveis, foi necessário fazer passar como atributo na TG, para além daqueles já referidos para as dependências de controlo, uma string contexto (herdado: String contexto_in, sintetizado: String contexto_out) para poder distinguir o contexto em que uma instrução é visitada e assim, se essa string contiver "CORPO_FUNCAO", podemos iniciar a procura de dependências, mas se contiver "WHILE", essa procura é adiada, tal como foi explicado na segunda abordagem. Também, para auxiliar a

segunda metodologia, uma lista de números de instrução (herdado: TreeSet<Integer> nrs_instrucao_while_in, sintetizado: TreeSet<Integer> nrs_instrucao_while_out) vai armazenar os números das instruções que são adiadas, para que no final do bloco estas possam ser alvo de uma procura de dependências. Finalmente, o último atributo (herdado: String bloco_if_in; sintetizado: String bloco_if_out) surge da necessidade de distinguir diferentes cenários em que uma dependência pode ser encontrada, estes cenários serão explicados posteriormente, mas estes dependem do bloco de instruções de uma estrutura if em que ambas as instruções envolvidas numa dependência se encontram. Este atributo toma o valor "THEN"quando o bloco then de um if é visitado e "ELSE"quando o bloco else é visitado.

O primeiro passo, para que a deteção de dependência de dados fosse possível, foi modificar a inserção de nodos no grafo, isto é, em termos de conteúdo. No CFG, tínhamos mencionado que da classe Instrucao apenas nos interessava a própria instrução em si. No PDG, isso já não se verifica e 4 novas variáveis foram adicionadas à classe. Para detetar uma dependência entre duas instruções é necessário sabermos que variáveis são referenciadas e também definidas, por isso, na classe Instrucao, foram criados dois HashSet<String> (variaveis_definidas e variaveis_referenciadas) para guardar exatamente essas variáveis. E também como já foi referido, é necessário saber o contexto em que uma instrução surge ("CORPO_FUNCAO"ou "WHILE") e se esta está num bloco then, else ou em nenhum ("THEN", "ELSE",), por isso, as outras duas variáveis adicionadas à classe foram String contexto e String bloco_if, respetivamente.

Dos 6 tipos de instrução existentes na linguagem Cmb, as instruções onde existe definição de variáveis são o scan (read) e a atribuição (atribuicao), e em todas as 6 é possível referenciar variáveis. Posto isto, para que na altura da inserção de um novo nodo no grafo, as variáveis definidas e referenciadas estivessem presentes, foi necessário sintetizá-las até cada uma das 6 instruções. De seguida mostramos a produção original da instrução atribuição para auxiliar a explicação.

Listing 13: Produção atribuicao

```
1 atribuicao
2         :     ^('=' ID expr)
3         ;
```

O caso da atribuição é o único caso em que simultaneamente, pode existir definição e referência de variáveis. Neste caso, ID é a variável definida e expr uma expressão onde podem existir variáveis referenciadas, é, então, necessário fazer chegar através de expr as variáveis referenciadas e por isso, o excerto seguinte ilustra a ação criada no SNT expr para que tal fosse possível.

Listing 14: Produção expr

```
1 expr returns [String instrucao, HashSet<String> vars_ref]
2     @init {
3         HashSet<String> vf = new HashSet<String>();
4     }
5         :    ^('||' a=expr b=expr)   {$expr.instrucao = $a.instrucao + "||" + $b.
              instrucao; vf = $a.vars_ref; vf.addAll($b.vars_ref); $expr.vars_ref = vf;
              }
6         ...
7         |    ^('!' a=expr) {$expr.instrucao = "!" + $a.instrucao; vf = $a.vars_ref;
              $expr.vars_ref = vf;}
8         |    factor {$expr.instrucao = $factor.instrucao;    $expr.vars_ref = $factor.
              vars_ref;}
9         ;
```

Podemos observar que o atributo vars_ref é sintetizado contendo as possíveis variáveis referenciadas. A produção do SNT factor não é aqui ilustrada, mas pode ser consultada em anexo.

Existe mais uma situação onde há definição de variáveis, esta é quando existem parâmetros de entrada numa função. Até agora, o cabeçalho de uma função tinha sido ignorado, mas agora é necessário capturar as variáveis que são passadas como parâmetro e adicioná-las à primeira instrução do grafo, aquela que representa a entrada na função. De seguida é apresentada a produção do SNT funcao, onde podemos observar que as variáveis definidas no cabeçalho são passadas como atributo herdado ao corpo da função.

Listing 15: Produção funcao

As produções que sintetizam esse atributo podem ser consultadas em anexo.

Em baixo, está a produção do SNT corpo_funcao, onde podemos observar a inserção do nodo inicial, assim como a passagem dos atributos já mencionados ao SNT statements.

Listing 16: Produção corpo funcao

```
corpo_funcao [GrafoPDG g_in, String id_funcao, HashSet<String> vars_def] returns [
    GrafoPDG g_out]
    @init{
        GrafoPDG g = $corpo_funcao.g_in;
        TreeSet<Integer> nrs = new TreeSet<Integer>();
        TreeSet<Integer> nrs_while = new TreeSet<Integer>();
        // 0 <=> Nodo inicial. E passado como parametro para que as restantes
            instrucoes se liguem a este
        int nr = g.putNodo(0, new Instrucao("ENTER (" + $corpo_funcao.id_funcao + ")"
            , $corpo_funcao.vars_def, null));
        nrs.add(nr);
    }
        : ^(CORPO declaracoes statements[$corpo_funcao.g_in, "CORPO_FUNCAO", nrs, "",
            nrs_while, ""])
        {
            $corpo_funcao.g_out = $statements.g_out;
        }
        ;
```

De seguida, é mostrada apenas uma produção statement, onde, após processar a instrução, neste caso atribuicao, o último nodo executado é conectado à atribuição juntamente com o atributo herdado label.

Listing 17: Produção corpo funcao

```
statement [GrafoPDG g_in, String contexto_in, TreeSet<Integer>
    nrs_ultima_instrucao_in, String label_in, TreeSet<Integer>
    nrs_instrucao_while_in, String bloco_if_in] returns [GrafoPDG g_out, String
    contexto_out, TreeSet<Integer> nrs_ultima_instrucao_out, String label_out,
    TreeSet<Integer> nrs_instrucao_while_out, String label_out, String bloco_if_out]
    @init {
        GrafoPDG g = $statement.g_in;
        String cx = $statement.contexto_in;
        String bi = $statement.bloco_if_in;
    }
        :   atribuicao[g, $statement.label_in, cx, nrs_instrucao_while_in, bi]
            {
                g = $atribuicao.g_out;

                // verifica se existem instrucoes anteriormente executadas e conecta
                    essas instrucoes Ã  nova instrucao
                g.checkAndPutCaminho($statement.nrs_ultima_instrucao_in, new
                    ParNrInstrucaoLabel($atribuicao.nrs_ultima_instrucao_out.first(),
                     $statement.label_in));

                $statement.g_out = g;
                $statement.contexto_out = cx;
                $statement.nrs_ultima_instrucao_out = $statement.
                    nrs_ultima_instrucao_in;
                $statement.label_out = $atribuicao.label_out;
                $statement.nrs_instrucao_while_out = $atribuicao.
                    nrs_instrucao_while_out;
                $statement.bloco_if_out = bi;
            }
        | ...
        ;
```

As restantes produções do SNT statement são semelhantes, à exceção do whiles, que adiciona um caminho extra, que é do predicado de controlo para si mesmo.

É ao nível dos SNT correspondentes às 6 instruções que a verificação de dependência de dados é feita. Vamos ilustrar 3 casos distintos para demonstrar a nossa implementação.

A próxima produção a ser exibida é a do SNT atribuicao, onde ocorre a inserção de um novo nodo no grafo, como já mencionámos anteriormente e também, onde ocorre a verificação de dependência de dados.

Listing 18: Produção atribuicao

```
1  atribuicao [GrafoPDG g_in, String label_in, String contexto_in, TreeSet<Integer>
       nrs_instrucao_while_in, String bloco_if_in] returns [GrafoPDG g_out, TreeSet<
       Integer> nrs_ultima_instrucao_out, String label_out, TreeSet<Integer>
       nrs_instrucao_while_out]
2      @init {
3          GrafoPDG g = $atribuicao.g_in;
4          HashSet<String> variaveis_definidas = new HashSet<String>();
5          String cx = $atribuicao.contexto_in;
6          TreeSet<Integer> nrs_instrucao_while = $atribuicao.nrs_instrucao_while_in;
7          String bi = $atribuicao.bloco_if_in;
8      }
9      :     ^('=' ID {variaveis_definidas.add($ID.text);} expr)
10     {
11         TreeSet<Integer> nrs = new TreeSet<Integer>();
12         // cria nodo no grafo e guarda o nr da instrucao
13         nrs.add(g.putNodo(new Instrucao($ID.text + " = " + $expr.instrucao,
               variaveis_definidas, $expr.vars_ref, cx, bi)));
14
15         // verifica existencia de dependencia de dados apenas se for uma instrucao
               que nao se encontre dentro de um if ou while (contexto "IF" ou "WHILE")
16         if (cx.equals("CORPO_FUNCAO")) {
17             g.checkDependenciasDados(nrs.first());
18         }
19         // se for uma instrucao que se encontre dentro de um while, entao o nr da
               instrucao e guardado para ser verificado no final do bloco while
20         if (cx.equals("WHILE")) {
21             nrs_instrucao_while.addAll(nrs);
22         }
23
24         $atribuicao.nrs_ultima_instrucao_out = nrs;
25         $atribuicao.nrs_instrucao_while_out = nrs_instrucao_while;
26         $atribuicao.g_out = g;
27         $atribuicao.label_out = $atribuicao.label_in;
28     }
29     ;
```

Tal como já foi dito, um nodo é inserido no grafo com mais informação, a instrução em si, as variáveis definidas, obtidas do token ID, as variáveis referenciadas, obtidas a partir do SNT expr, e o contexto e o bloco_if obtidos dos atributos herdados pelo SNT atribuicao. Depois, utilizando o atributo herdado contexto_in, se esta atribuição surgir no contexto do corpo da função então verifica a existência de dependências de dados para essa instrução, fazendo uma chamada ao método checkDependenciasDados da instância do grafo passada como atributo herdado. Caso a atribuição surja dentro de um bloco while, então não é feita a verificação e o número da instrução do while é guardado na variável nrs_instrucao_while que será sintetizada no final da execução desta produção.

A produção seguinte é a do SNT ifs, que apenas difere da atribuicao pela existência dos blocos then e else.

Listing 19: Produção ifs

```
1  ifs [GrafoPDG g_in, String contexto_in, TreeSet<Integer> nrs_ultima_instrucao_in,
       String label_in, TreeSet<Integer> nrs_instrucao_while_in] returns [GrafoPDG
       g_out, TreeSet<Integer> nrs_ultima_instrucao_out, String label_out, TreeSet<
       Integer> nrs_instrucao_while_out]
```

```
@init {
    GrafoPDG g = $ifs.g_in;
    int nr_ult_inst_exp = -1;
    TreeSet<Integer> nrs_exp = new TreeSet<Integer>();
    TreeSet<Integer> nrs_instrucao_while = $ifs.nrs_instrucao_while_in;
    String cx = $ifs.contexto_in;
}
    :   ^(IF expr
            {
                nr_ult_inst_exp = g.putNodo(new Instrucao($IF.text + "(" + $expr.
                    instrucao + ")", null, $expr.vars_ref, cx));

                nrs_exp.add(nr_ult_inst_exp);

                if (cx.equals("CORPO_FUNCAO")) {
                    g.checkDependenciasDados(nrs_exp.first());
                }
                if (cx.equals("WHILE")) {
                    nrs_instrucao_while.addAll(nrs_exp);
                }

            }
            a=bloco[g, cx, nrs_exp, "T", nrs_instrucao_while, "THEN"]
            {
                g = $a.g_out;
                if (cx.equals("WHILE")) {
                    nrs_instrucao_while.addAll($a.nrs_instrucao_while_out);
                }
            }
            (b=bloco[g, cx, nrs_exp, "F", nrs_instrucao_while, "ELSE"]
            {
                g = $b.g_out;
                if (cx.equals("WHILE")) {
                    nrs_instrucao_while.addAll($b.nrs_instrucao_while_out);
                }
            } )?
        )
        {
            $ifs.nrs_ultima_instrucao_out = nrs_exp;
            $ifs.nrs_instrucao_while_out = nrs_instrucao_while;
            $ifs.g_out = g;
            $ifs.label_out = $ifs.label_in;
        }
    ;
```

As ações executadas após o predicado de controlo são as mesmas que para a atribuição. Ao o bloco de instruções then são passados os mesmo atributos que foram passados em todas as produções que mostrámos, mas com valores diferentes, o número da última instrução executada é o número do predicado de controlo (expressão) para que as instruções do bloco se conectem a essa instrução, a label é passada com o valor "T"para associar aos ramos das conexões que serão criadas, e o bloco_if com o valor "THEN"para identificar as instruções como pertencentes a um bloco then. Após a execução do bloco, se as instruções do if surgirem no contexto de um while então os números que foram guardados no bloco são adicionados a uma variável que será depois sintetizada no final da produção. O bloco else é idêntico, só mudam os valores da label para "F"e o bloco_if para "ELSE".

Finalmente, a última produção é a do SNT whiles, onde no final da execução dos SNT do lado direito da produção, caso o ciclo while tenha surgido no contexto do corpo da função (isto é, se existirem whiles aninhados, todas as instruções internas são adiadas até ao final do mais externo), é feita a verificação da existência de dependências de dados para as instruções que foram adiadas.

Listing 20: Produção whiles

```
whiles [GrafoPDG g_in, String contexto_in, TreeSet<Integer> nrs_ultima_instrucao_in,
    String label_in, TreeSet<Integer> nrs_instrucao_while_in, String bloco_if_in]
    returns [GrafoPDG g_out, TreeSet<Integer> nrs_ultima_instrucao_out, String
    label_out, TreeSet<Integer> nrs_instrucao_while_out]
@init {
    GrafoPDG g = $whiles.g_in;
    int nr_ult_inst_exp = -1;
    TreeSet<Integer> nrs_exp = new TreeSet<Integer>();
    TreeSet<Integer> nrs_instrucao_while = $whiles.nrs_instrucao_while_in;
    String cx = $whiles.contexto_in;
    String bi = $whiles.bloco_if_in;
}
    :    ^(WHILE expr
            {
                nr_ult_inst_exp = g.putNodo(new Instrucao($WHILE.text + "(" + $expr.
                    instrucao + ")", null, $expr.vars_ref, "WHILE", bi));
                nrs_exp.add(nr_ult_inst_exp);
                nrs_instrucao_while.addAll(nrs_exp);
            }
            bloco[g, "WHILE", nrs_exp, "T" ,nrs_instrucao_while, bi]
            {
                g = $bloco.g_out;
                nrs_instrucao_while.addAll($bloco.nrs_instrucao_while_out);
            })
            {
                if (cx.equals("CORPO_FUNCAO")) {
                    g.checkDependenciasDadosWhile(nrs_instrucao_while);
                    nrs_instrucao_while.clear();
                }

                $whiles.nrs_ultima_instrucao_out = nrs_exp;
                $whiles.nrs_instrucao_while_out = nrs_instrucao_while;
                $whiles.g_out = g;
                $whiles.label_out = $whiles.label_in;
            }
    ;
```

Nesta produção a verificação de dependências de todas as instruções pertencentes ao while, inclusive o predicado de controlo são adiadas até ao final da produção. Ao bloco de instruções é passado o atributo contexto_in com o valor ”WHILE”, que obriga ao adiamento da verificação de dependências das instruções no bloco, o atributo nrs_ultima_instrucao_in com o número de instrução do predicado de controlo (expressão) e o atributo label_in com o valor ”T”(todas as instruções de um bloco while têm no caminho criado do predicado de controlo para si a label ”T”). Por fim, a verificação de dependências de dados é feita fazendo uma chamada ao método checkDependenciasDadosWhile da instância do grafo.

A Tree Grammar completa pode ser consultada em anexo.

A explicação dos métodos de verificação de existência de dependência de dados mencionados nesta secção será feita na secção seguinte.

Deteção de dependências de dados

Nesta secção explicaremos os métodos de deteção de dependências de dados criados. Tal como foi dito na secção anterior, existem dois métodos que são chamados na Tree Grammar, um método (checkDependenciasDados) que deteta as dependências para instruções fora de um ciclo while e um método (checkDependenciasDadosWhile) para instruções dentro de um ciclo while.

Foram identificados vários cenários que permitem a deteção de dependências apenas em determinadas circunstâncias. Estes cenários representam uma possível dependência entre duas instruções A e B, sendo que uma variável referenciada em B pode depender de uma definida em A.

Os cenários são:

1. B encontra-se fora de uma estrutura if, ou seja, não se encontra num bloco then ou else
2. B pertence a um bloco then
3. B pertence a um bloco else

Os cenários mencionados são conjugados com um dos cenários seguintes:

4. A pertence ao corpo da função (não pertence a nenhum tipo de bloco)
5. A pertence a um ciclo while
6. A pertence a um bloco then
7. A pertence a um bloco else

O método checkDependenciasDados recebe como parâmetro o número de instrução de B, e para cada variável referenciada em B inicia a procura de dependências de dados percorrendo as instruções precedentes, para isso invocando o método com o mesmo identificador mas com variáveis diferentes (princípio do polimorfismo presente no Java e noutras linguagens orientadas a objetos). Estas variáveis são, pela ordem respetiva, o número da instrução cuja variável definida está a ser comparada, o número da instrução sobre a qual se está a procurar dependências, a variável sobre a qual se está a procurar dependências e uma variável de controlo. A implementação em Java é ilustrada a seguir:

Listing 21: Método checkDependenciasDados

```
1  public void checkDependenciasDados(int nr_instrucao){
2          Instrucao i = this.getNodos().get(nr_instrucao);
3
4          // para cada variavel referenciada na instrucao passada por
               parametro
5          for (String var_ref : i.getVariaveis_referenciadas()){
6              boolean dep_encontrada = false;
7              boolean procurar_fora_bloco = false;
8              // inicia a procura a partir da instrucao imediatamente anterior
                   Ã  instrucao nr_instrucao
9              int inst_comp = nr_instrucao-1;
10             // procura enquanto nenhuma dependencia for encontrada, ou ate
                   todas as instrucoes da funcao tiverem sido comparadas
11             while (!dep_encontrada && inst_comp>=0) {
12                 ParDependenciaInstrucao p = checkDependenciasDados(inst_comp
                       , nr_instrucao, var_ref, procurar_fora_bloco);
13                 dep_encontrada = p.isExiste_dependencia();
14                 inst_comp = p.getInstrucao_seguinte();
15                 procurar_fora_bloco = p.isProcurar_fora_bloco();
16             }
17         }
18     }
```

E o método de deteção de dependência de dados entre duas instruções, com base nos cenários mencionados, fica com a seguinte estrutura (apenas mostramos a estrutura base, não pusemos o código completo):

Listing 22: Método ParDependenciaInstrucao

```
1  public ParDependenciaInstrucao checkDependenciasDados(int nodo_anterior, int
       nodo_posterior, String var_ref, boolean procurar_fora_bloco){
2      Instrucao i = this.getNodos().get(nodo_anterior);
3      Instrucao i_post = this.getNodos().get(nodo_posterior);
4
5
```

```
6       // CENARIO - 1
7       if ((!i_post.getBloco_if().equals("THEN") && !i_post.getBloco_if().
            equals("ELSE")) || procurar_fora_bloco) {
8           // CENARIO - 5
9           if (i.getContexto().equals("WHILE")){ ... }
10          // CENARIO - 4
11          else if (!i.getBloco_if().equals("THEN") && !i.getBloco_if().equals(
                "ELSE")) { ... }
12          // CENARIO - 7
13          else if (i.getBloco_if().equals("ELSE")) { ... }
14          // CENARIO - 6
15          else if (i.getBloco_if().equals("THEN")) { ... }
16      }
17      // CENARIO - 3
18      else if (i_post.getBloco_if().equals("ELSE")) { ... }
19      // CENARIO - 2
20      else if (i_post.getBloco_if().equals("THEN")) { ... }
21  }
```

De seguida, para cada cenário vamos explicitar o que acontece, deixando os detalhes de implementação para consulta do código em anexo.

No cenário 5 (A pertence a um ciclo while) são procuradas dependências nas instruções pertencentes ao bloco while, parando a procura quando uma dependência for encontrada e adicionado-a ao grafo. Uma dependência encontrada apenas num bloco else não implica que a procura pare porque é ainda possível encontrar uma dependência em instruções anteriores, por isso é também feita uma procura no bloco then. Se uma dependência for encontrada tanto no bloco then como no bloco else, então a procura para, porque qualquer instrução anterior à estrutura if não influenciará B, caso contrário continua a procura (este comportamento é o que acontece no cenário 7). Caso se encontre uma dependência apenas no bloco then, a procura não para (este comportamento é o que acontece no cenário 6).

Independentemente de uma dependência ser encontrada dentro do bloco while, também é iniciada uma procura de dependências a partir da instrução imediatamente anterior ao predicado de controlo do ciclo while.

O cenário 4 (A pertence ao corpo da função (não pertence a nenhum tipo de bloco)) é o mais simples, pois se uma dependência for encontrada esta é inserida no grafo e a procura para.

Os cenários 6 (A pertence a um bloco then) e 7 (A pertence a um bloco else) já foram explicados no cenário 5.

No cenário 3 (B pertence a um bloco else), como a instrução B pertence a um bloco else, nunca poderá depender de uma instrução pertencente ao bloco then desse mesmo if. Por isso, se uma dependência for encontrada dentro do bloco a procura para e a dependência é inserida no grafo, caso contrário a procura continua a partir do predicado de controlo desse if.

O cenário 2 (B pertence a um bloco then) é semelhante ao 3, ou seja, se uma dependência for encontrada no interior do bloco then então a procura para e a dependência é inserida no grafo, caso contrário a procura continua a partir do predicado de controlo desse if.

O método checkDependenciasDadosWhile, ou seja, o método que deteta as dependências para instruções dentro de um ciclo while, recebe como parâmetro um TreeSet<Integer> com os números de instrução contidos no ciclo while incluindo o predicado de controlo. Esse TreeSet é percorrido por ordem decrescente, ou seja, começa da última instrução e termina no predicado de controlo. Vamos considerar B o número de instrução em cada iteração. Para cada variável referenciada em B é iniciada uma procura de dependências de dados, comparando B com todas as instruções presentes no ciclo, iniciando a procura no próprio B e avançando para instruções anteriores. Quando o predicado de controlo for atingido e ainda houver instruções para procurar, então a procura continua na última instrução do ciclo e para na instrução imediatamente a seguir a B. Quando uma dependência for encontrada a procura para e inicia uma nova procura para as restantes variáveis referenciadas em B. Se não houver mais variáveis então avança para a instrução anterior a B e assim sucessivamente até que todas as instruções tenham sido verificadas.

À semelhança do método anterior, este também possui uma estratégia baseada nos cenários apresentados anteriormente. Por isso, a estrutura do método é apresentada de seguida.

Listing 23: Método ParDependenciaInstrucao

```
public void checkDependenciasDadosWhile(TreeSet<Integer> nrs_instrucao_while
    ){
     for (int inst_dependente : nrs_instrucao_while.descendingSet()){
         Instrucao i = this.getNodos().get(inst_dependente);
         for (String var_ref : i.getVariaveis_referenciadas()){
             int inst_comp = inst_dependente;
             while(nr_inst_comparadas <= nr_inst_a_comparar && !
                dep_encontrada){
                 Instrucao i_comp = this.getNodos().get(inst_comp);

                 // CENARIO 1
                 if ((!i.getBloco_if().equals("THEN") && !i.getBloco_if().
                    equals("ELSE")) || proc_fora_bloco) {
                     // CENARIO 4
                     if (!i_comp.getBloco_if().equals("THEN") && !i_comp.
                        getBloco_if().equals("ELSE")) { ... }
                     // CENARIO 7
                     else if (i_comp.getBloco_if().equals("ELSE")) { ... }
                     // CENARIO 6
                     else if (i_comp.getBloco_if().equals("THEN")) { ... }
                 }
                 // CENARIO 3
                 else if (i.getBloco_if().equals("ELSE")) { ... }
                 // CENARIO 2
                 else if (i.getBloco_if().equals("THEN")) { ... }

                 // atualiza inst_comp
                 ...
             }
             // insere possiveis dependencias encontradas na procura, mas que
                 so podem ser confirmadas no final da procura
             ...

             // inicia a procura a partir da instrucao imediatamente anterior
                 a expressao do while
             ...
         }
     }
}
```

Pode-se observar a iteração sobre todas as instruções do ciclo while (1º for), a iteração sobre as variáveis referenciadas em B (2º for) e a procura de dependências de dados percorrendo as instruções iniciando em B (while).

De seguida, para cada cenário vamos explicitar o que acontece, deixando os detalhes de implementação para consulta do código em anexo.

O cenário 4 (A pertence ao corpo da função (não pertence a nenhum tipo de bloco)) não é tão simples como no método anterior, porque existem algumas exceções que precisam de se ter em conta. A primeira exceção é a possibilidade de a instrução poder depender dela própria, no entanto, isto só é possível se mais nenhuma dependência para além dessa tiver sido encontrada, por isso, a dependência não é inserida de imediato e a sua inserção é adiada para o final da verificação das dependências para B. Outra exceção depende da ordem das instruções, ou seja, se o número de instrução de A for menor que B então não há necessidade de procurar mais, nem dentro nem fora do while e por isso uma flag é ativada para que tal procura fora do while seja ou não seja

realizada. Finalmente, se nenhuma destas exceções se verificar, e uma dependência for encontrada então esta é inserida no grafo e a procura para.

No cenário 7 (A pertence a um bloco else) é necessário a procura tanto no bloco else como no then, já que, por ser um ciclo, uma instrução pode depender tanto de um bloco como do outro. Uma dependência encontrada num bloco é imediatamente inserida no grafo. Se em ambos os blocos forem detetadas dependências então a procura para. Neste cenário também existe a exceção da ordem das instruções, ou seja, a dependência encontrada, quer a instrução seja no bloco then ou else, se for anterior a B então a procura para.

No cenário 6 (A pertence a um bloco then) a procura é feita no bloco then e se for encontrada uma dependência esta é inserida no grafo. Quer encontre uma dependência ou não, a procura continua nas instruções imediatamente anteriores.

No cenário 3 (B pertence a um bloco else), se uma dependência for encontrada dentro do bloco else, então a procura para. Se uma dependência for encontrada no bloco then, esta não é inserida imediatamente porque se existir uma dependência fora do if, essa dependência entre os blocos deixa de fazer sentido. Existe uma exceção neste cenário que dita que pode existir uma dependência de B para B, apenas se nenhuma dependência for encontrada no bloco then.

No cenário 2 (B pertence a um bloco then) se uma dependência for encontrada no interior do bloco then então a procura para e a dependência é inserida no grafo, caso contrário a procura continua a partir do predicado de controlo desse if. Se a dependência encontrada for uma dependência dela própria, esta não é inserida de imediato, aguardando até ao final da procura no ciclo para garantir que outra dependência não foi encontrada.

As classes Java podem ser consultadas em anexo.

Exemplo de um grafo PDG gerado:

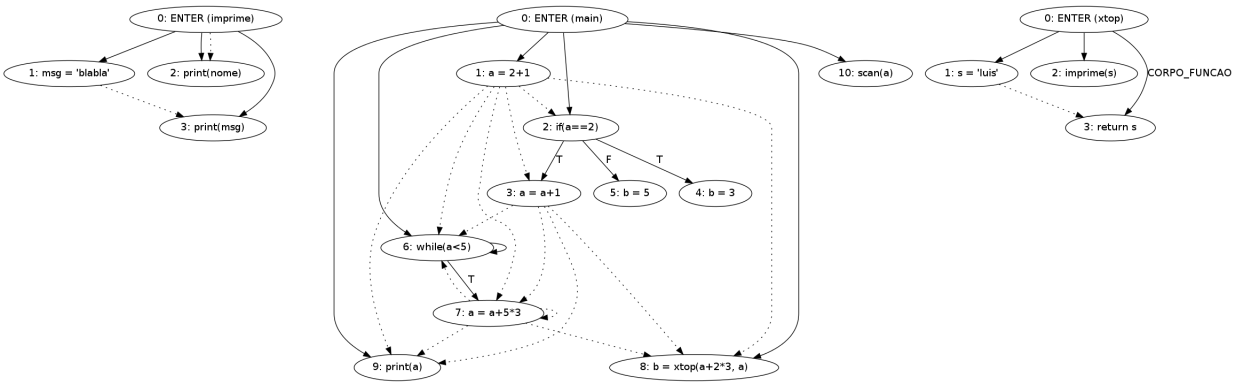

Figura 2: Grafo PDG

4.2.3 System Dependency Graph

Um System Dependecy Graph é um representação de um programa com uma estrutura idêntica ao PDG mas com outra informação. Neste tipo de grafos temos a representado o fluxo das instruções, as dependências de dados, as chamadas de funções e a passagem de parâmetros.

Aqui as verificações que são realizadas são geralmente na passagem de parametros quando ocorre a invocação de outras funções do sistema. A optimização pode ocorrer, por exemplo, quando a invocação de uma função é feita sempre com os mesmos parâmetros.

Para a implementação deste modulo apoiamo-nos na estrutura já criada para o PDG adicionando alguns atributos á gramática e criando novas classes para guardar a informação.

É essencial guardar o cabeçalho de uma função para sabermos que argumentos são esperados e o valor que será retornado. O atributo `TreeMap<String, CabecalhoFuncao> funcoes_out` que existe na produção `funcao` da gramática é responsável por armazenar tudo o que precisamos de saber para invocar uma função. O `TreeMap<String, CabecalhoFuncao>` tem como chave o nome da função (logo partimos do principio que não

há funções com o mesmo nome) e o valor é uma instância de `CabecalhoFuncao`, esta classe guarda o nome da função, o tipo de retorno e ainda os parametros com a respectiva ordem.

É na produção `cabeçalho` que se consegue obter o que necessitamos:

Listing 24: Produção cabecalho

```
cabecalho [GrafoSDG g_in] returns [GrafoSDG g_out, String id, HashSet<String>
    vars_def, String nomeFuncao_out]
@init{
    HashSet<String> variaveis_definidas = new HashSet<String>();
    TreeMap<Integer, String> parms = new TreeMap<Integer, String>();
}
    :  ^(CAEBECALHO tipo ID (argumentos{variaveis_definidas = $argumentos.vars_def;
       parms = $argumentos.parms_out; })?)
    {
        $cabecalho.id = $ID.text;
        $cabecalho.vars_def = variaveis_definidas;

        // cria uma instancia para guardar a informacao do cabecalho da funcao
        CabecalhoFuncao cf = new CabecalhoFuncao();
        cf.setNomeFuncao($ID.text);
        cf.setTipoRetorno($tipo.text_out);
        if($argumentos.parms_out != null){
            cf.setParametros($argumentos.parms_out);
        }
        // adiciona o cf ao GrafoSDG
        $cabecalho.g_in.putCabecalhoFuncao(cf);
        $cabecalho.g_out = $cabecalho.g_in;
    }
    ;
```

É na produção `argumentos` que se obtêm os parametros e se calcula a ordem pela qual aparecem.

Listing 25: Produção argumentos

```
argumentos returns [HashSet<String> vars_def, TreeMap<Integer, String> parms_out]
@init{
    HashSet<String> variaveis_definidas = new HashSet<String>();
    // variavel para guardar os parametros
    TreeMap<Integer, String> parms = new TreeMap<Integer, String>();
    // inteiro que indica a ordem do parametro
    Integer i = 0;
}
    :  ^(ARGUMENTOS (declaracao { variaveis_definidas.add($declaracao.var_def); parms
       .put(i, $declaracao.var_def); i = i+1;}
    )+)
    { $argumentos.vars_def = variaveis_definidas;
      // sintetiza o TreeMap com os parametros
      $argumentos.parms_out = parms;
    }
    ;
```

A classe GrafoSDG agora utilizada, é um `extends` da classe GrafoPDG em que se acrescentou as variáveis de instância necessárias para guardar as invocações e os cabecalhos das funções.

Já recolhemos a estrutura das funções, agora precisamos de ver quando ocorre uma invocação e arranjar uma forma para fazer um *mapping* entre o cabecalho e a função.

A estratégia foi criar uma classe chamada `ChamadasFuncao` onde é registado o número da instrução, o nome da função invocada, as variáveis que podem ser retornadas e ainda os parametros com a respectiva ordem. Através da classe `CabecalhoFuncao` e `CahamdasFuncao` e tendo em conta as suas variáveis de instância, verificamos que é agora possível fazer um *mapping* entre as invocações de um função com a respectiva função. De seguida é apresentada a forma utilizada para detectar as invocações.

Como não poderia deixar de ser, na própria produção `invocacao` é registada a chamada a uma função, no entanto não é possível registar tudo aqui, isto porque se a invocação não for uma instrução isolada (ou seja, se estiver na condição de um if ou de while por exemplo) temos de sintetizar alguma informação recolhida neste ponto até à produção onde jé temos tudo o que é necessário.

Listing 26: Produção instrução

```
invocacao [GrafoSDG g_in, String contexto_in, String label_in, TreeSet<Integer>
    nrs_instrucao_while_in, String bloco_if_in] returns [GrafoSDG g_out, TreeSet<
    Integer> nrs_ultima_instrucao_out, String instrucao, String label_out, HashSet<
    String> vars_ref, TreeSet<Integer> nrs_instrucao_while_out, ChamadasFuncao
    cFuncao_out]
@init {
    GrafoSDG g = $invocacao.g_in;
    HashSet<String> variaveis_referenciadas = new HashSet<String>();
    String cx = $invocacao.contexto_in;
    TreeSet<Integer> nrs_instrucao_while = $invocacao.nrs_instrucao_while_in;
    String bi = $invocacao.bloco_if_in;
}
    :  ^(INVOCACAO ID (args
        {
            variaveis_referenciadas =  $args.vars_ref;
        }
        )?)
    {
        if (cx.equals("FACTOR")) {
            $invocacao.instrucao = $ID.text + "(" + $args.ags + ")";
            $invocacao.vars_ref = variaveis_referenciadas;

            ChamadasFuncao cf_t = new ChamadasFuncao();
            cf_t.setNomeFuncao($ID.text);
            cf_t.setParametros($args.parametros_out);

            $invocacao.cFuncao_out = cf_t;
        }
        else {
            TreeSet<Integer> nrs = new TreeSet<Integer>();
            // cria nodo no grafo e guarda o nr da instrucao
            Integer numero_instrucao = g.putNodo(new Instrucao($ID.text + "(" + $args
                .ags + ")", null, variaveis_referenciadas, cx, bi));
            nrs.add(numero_instrucao);

            ChamadasFuncao cf = new ChamadasFuncao();
            cf.setNrInstrucao(numero_instrucao);
            cf.setNomeFuncao($ID.text);
            cf.setParametros($args.parametros_out);
            g.putChamadaFuncao(numero_instrucao, cf);

            $invocacao.cFuncao_out = null; // jÃ¡ estÃ¡ adicionado

            ...

            $invocacao.nrs_ultima_instrucao_out = nrs;
            $invocacao.nrs_instrucao_while_out = nrs_instrucao_while;
            $invocacao.g_out = g;
            $invocacao.label_out = $invocacao.label_in;
        }
    }
    ;
```

Caso a invocação venha do contexto "FACTOR"(vai entrar no Then do IF) é necessário sintetizar a instância da classe `ChamadasFuncao` para uma produção onde se consiga saber o número da instrução onde ocorre.

Mesmo em `factor` não se consegue saber, por isso continua a sintetizar a chamada da funcao (CF):

Listing 27: Produção funcao

```
factor returns [String instrucao, HashSet<String> vars_ref, ChamadasFuncao
    cFuncao_out]
@init {
    HashSet<String> variaveis_ref = null;
}
    ...
    | invocacao[null, "FACTOR", "", null, ""]
    {
        $factor.instrucao = $invocacao.instrucao;
        $factor.vars_ref= $invocacao.vars_ref;
        $factor.cFuncao_out = $invocacao.cFuncao_out;
    }
    ;
```

O mesmo acontece na `expr`:

Listing 28: Produção expr

```
expr returns [String instrucao, HashSet<String> vars_ref, ChamadasFuncao cFuncao_out]
@init {
    HashSet<String> vf = new HashSet<String>();
}
    ...
    |   factor {$expr.instrucao = $factor.instrucao; $expr.vars_ref = $factor.
        vars_ref; $expr.cFuncao_out = $factor.cFuncao_out; }
    ;
```

Depois da produção `expr` sintetizar a CF, vamos estar perante outras produção em que já é possível obter o número da instrução. Essas produções são: atribuição, write, ifs e whiles, só não é o read porque como é óbvio não é possível ler do standard input para uma função. Nas produções referidas existe em todas algo como:

Listing 29: Produção expr

```
//  caso exista vai adicionar uma invocacao a uma funcao
if($expr.cFuncao_out != null){
    $expr.cFuncao_out.setNrInstrucao(nr_ult_inst_exp);
    g.putChamadaFuncao(nr_ult_inst_exp, $expr.cFuncao_out);
}
```

A partir deste momento já temos registado todas as funções e todas as invocações que existem, o que se seguiu vou apenas criar um método que recebesse uma instância da classe GrafoSDG e gerar linguagem Dot para desenhar um grafo que representasse graficamente o resultado.

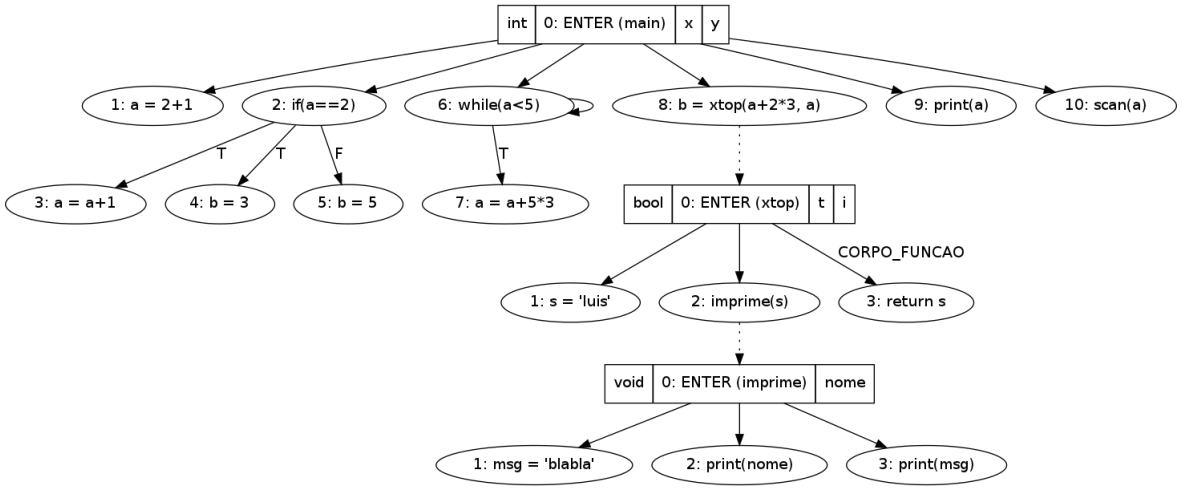

Figura 3: Grafo SDG

Analisando o grafo gerado, podemos verificar que algumas ligações não estão representadas mas estãom implititas, no caso dos argumentos, visto que foram colocados pela mesma ordem não achamos necessário estar a colocar. No entanto a ligação do "return"para a variavel que está a invocar a função é que não está implementada, falta concluir este pormenor.

4.2.4 Single Static Assignment

Single Static Assignment (SSA) é uma propriedade de ua representação intermédia (RI), que diz que cada variável é definida uma vez. AS variáveis que já existem na RI original são divididas em versões, em que as novas variáveis são normalmente representadas pelo nome original seguida de um digito. Mostrando alguns exemplos verificamos que os benificios são relativamente simples de detectar.

Se tivermos definido:

```
a = 3
a = 14
b = a
```

Á primeira vista reparamos logo que a primeira atribuição é desnecessária, se passarmos o código para o formato SSA ele ficará:

```
a1 = 3
a2 = 14
b1 = a2
```

Se estivermos perante um bom compilador, ele agora irá detectar que a primeira instrução não é necessária e que o "b"irá tomar o valor de "a2".

A ideia inicial era criar um texto igual ao de entrada mas no formato SSA, no entando o que nós fizemos foi um grafo como o CFG mas com as instruções no formato SSA.

Para resolver o problema criamos a classe `GrafoTGSSA` que é uma extensão da classe `Grafo`, nesta classe temos definidos:

Listing 30: Classe GrafoTGSSA

```
1 // variavel => versao
2 private TreeMap<String, Integer> versoesVariaveis;
3
```

```
// onde esta guardado o contexto onde as versoes das variaveis foram encontradas
// a1 => If_THEN; a2 => IF_ELSE
private TreeMap<String, String>  contextoVariaveis;
```

em `versoesVariaveis` guardamos a variável com o número da última versão que foi definida, enquanto que na variável de instância `contextoVariavel` fica guardado a variável com a respectiva versão e o contexto em que ela foi definida. Isto de ter o contexto é necessário por exemplo para o caso em que a seguir a um "IF"temos uma variável que está a ser definida no "THEN"ou no "ELSE", nesta situação o que estiver a seguir ao "IF"não vai saber de onde é que vem o valor calculado, pode ser de um dos 2.

Para resolver o problema começamos por identificar onde é que as variáveis podem ser definidas e chegamos à conclusão que podem em: read e atribuição. Logo, é nestas produções que a versão de uma variável vai ser alterada:

Listing 31: Produção atribuicao e read

```
atribuicao [GrafoTGSSA g_in, String label_in, String contexto_in] returns [GrafoTGSSA g_out, TreeSet<Integer> nrs_ultima_instrucao_out, String label_out]
@init {
    GrafoTGSSA g = $atribuicao.g_in;
}
    :    ^('=' ID expr[g, $atribuicao.contexto_in])
    {
        TreeSet<Integer> nrs = new TreeSet<Integer>();
        // cria nodo no grafo e guarda o nr da instrucao
        Integer i_id = $atribuicao.g_in.getVersaoVariavelNext($ID.text);
        nrs.add(g.putNodo(new Instrucao($ID.text + " = " + $expr.instrucao, null, null, "", "", $ID.text + i_id + " = " + $expr.instrucaoVersao)));

        //como ha uma atribuicao, a versao da variavel vai ter de ser incrementada
        Integer i = g.incrementaVariavel($ID.text);

        //vai adicionar a variavel e dizer qual e o contexto onde se encontra
        g.adicionaVariavelContexto($ID.text + i, $atribuicao.contexto_in);
        //System.out.println(g_in.getContextoVariaveis());

        ...
    }
    ;


read [GrafoTGSSA g_in, String label_in, String contexto_in] returns [GrafoTGSSA g_out, TreeSet<Integer> nrs_ultima_instrucao_out, String label_out]
@init {
    GrafoTGSSA g = $read.g_in;
}
    : ^(READ ID)
    {
        TreeSet<Integer> nrs = new TreeSet<Integer>();
        // cria nodo no grafo e guarda o nr da instrucao
        nrs.add(g.putNodo(new Instrucao($READ.text + "(" + $ID.text + ")", null, null)));

        //como ha uma atribuicao, a versao da variavel vai ter de ser incrementada
        Integer i = g.incrementaVariavel($ID.text);

        // vai adicionar a variavel e dizer qual e o contexto onde se encontra
        g.adicionaVariavelContexto($ID.text + i, $read.contexto_in);
        ...
    }
```

```
42      ;
```

Quando se chama o método incrementaVariavel(ID.text) ele vai automaticamente incremetar no TreeMap versoesVariaveis a versão daquela variável e devolve o valor da actual versão. De seguida adicionamos à estrutura onde estão os contextos o que nos interessa para no final conseguirmos fazer todas as verificações necessárias (adicionaVariavelContexto($ID.text + i$,atribuicao.contexto_in)).

Com toda a informação guardada que é necessária, bastou criar um método que calcula-se quais as dependências que podiam existir para uma variável, esse método está definido na classe GrafoTGSSA e chama-se getVersaoVariavelDependentes, recebe como parametro uma variável e vai verificar no que já está definido e tendo em conta o contexto as variáveis das quais o seu valor pode depender.

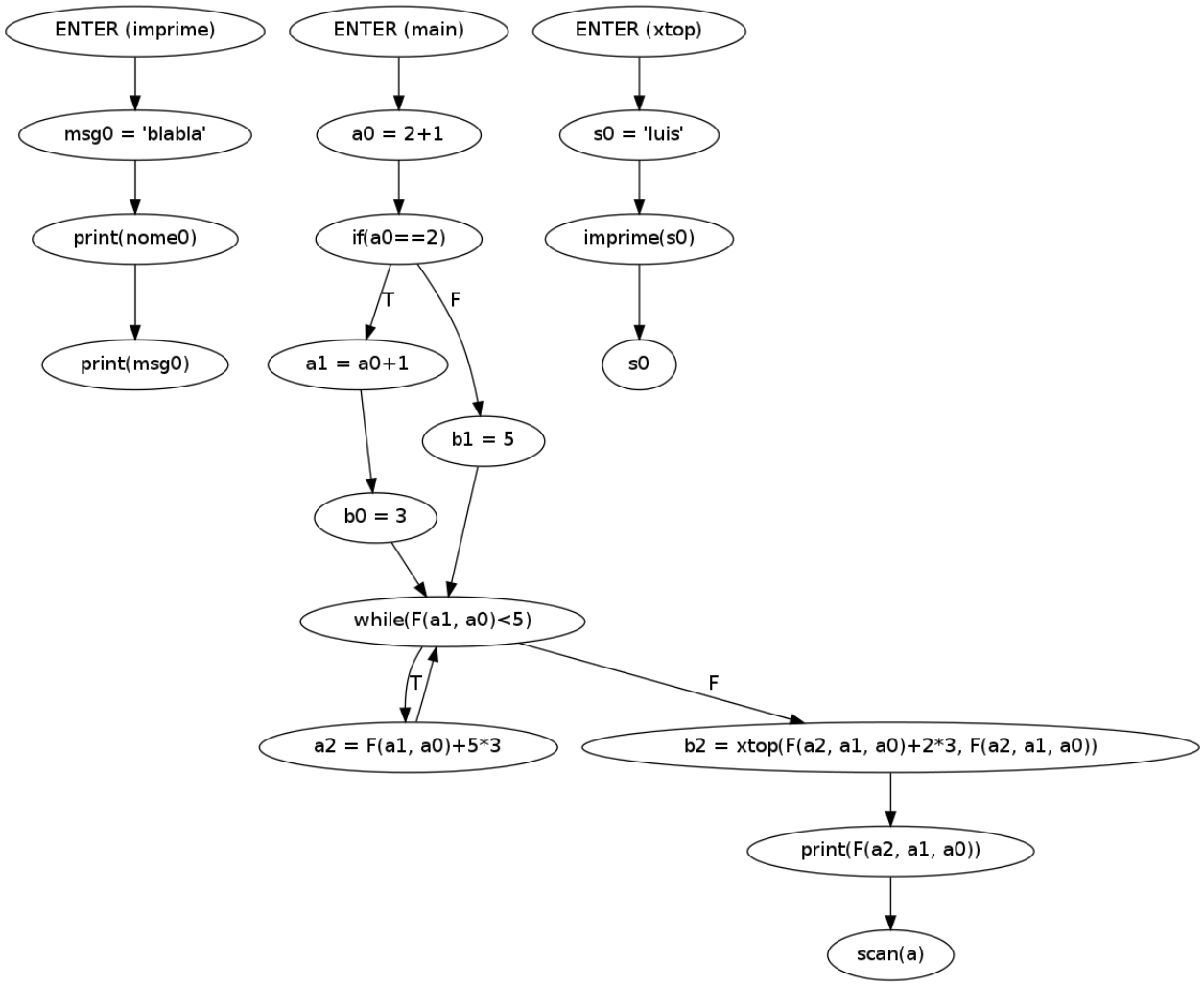

Figura 4: Grafo SSA

Nota: No grafo a letra ”F”dentro dos nodos representa o símbolo phi.

5 WebSite

Foi dada a sugestão de criar um pequeno site com páginas para mostrar os resultados, o que nós fizemos foi colocar um formulário para submeter o código que é pretendido analisar e depois o resultado é mostrado nas correspondentes opções do menu.

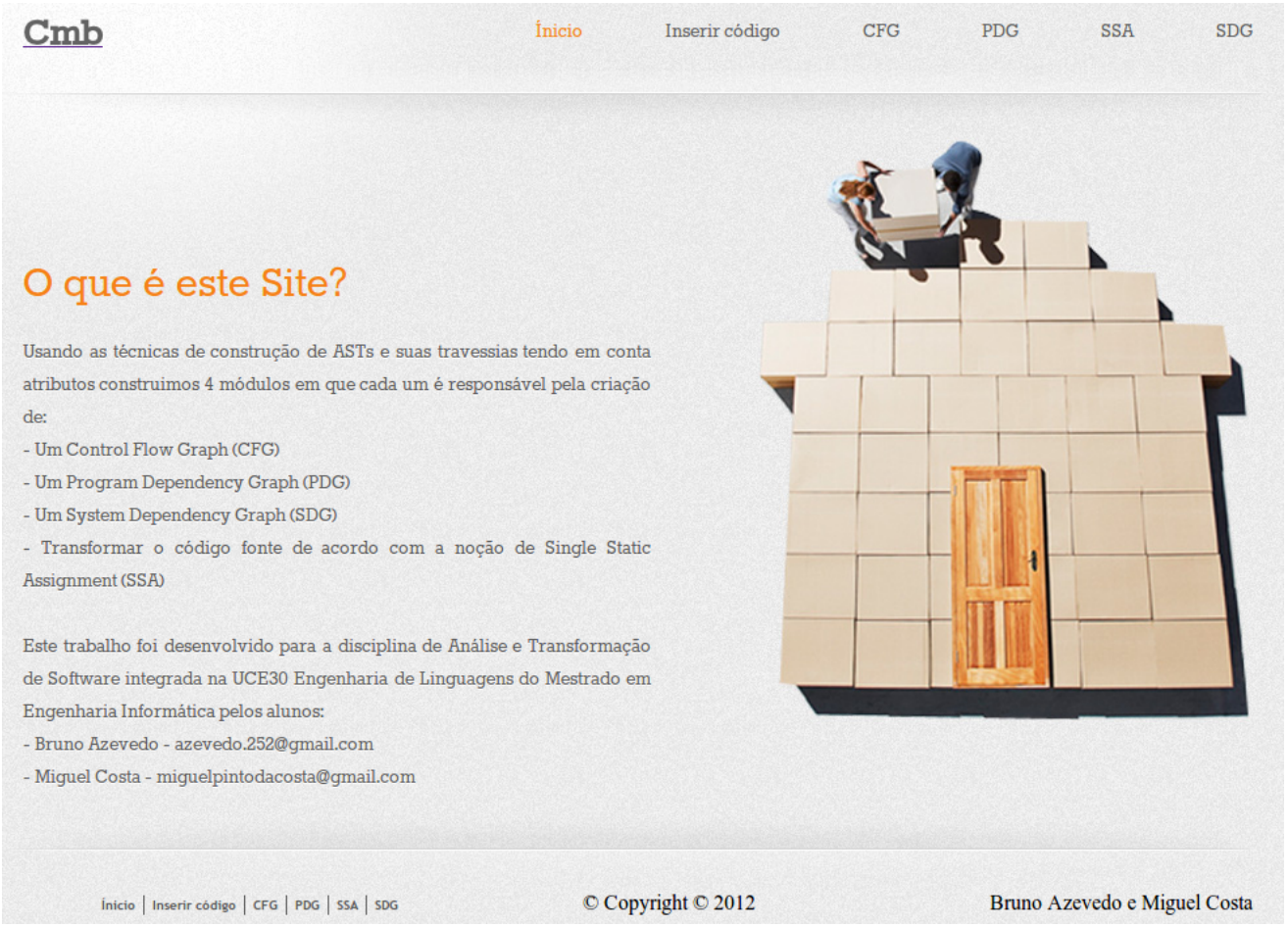

Figura 5: Página inicial

Esta é a página inicial apenas, em baixo aparece a página para se submeter o código a analisar:

Figura 6: Página de submissão do código

Depois de submetido o código, é possível ver os resultados que se pretende escolhendo CFG, PDG, SSA ou SDG nas categorias disponiveis.

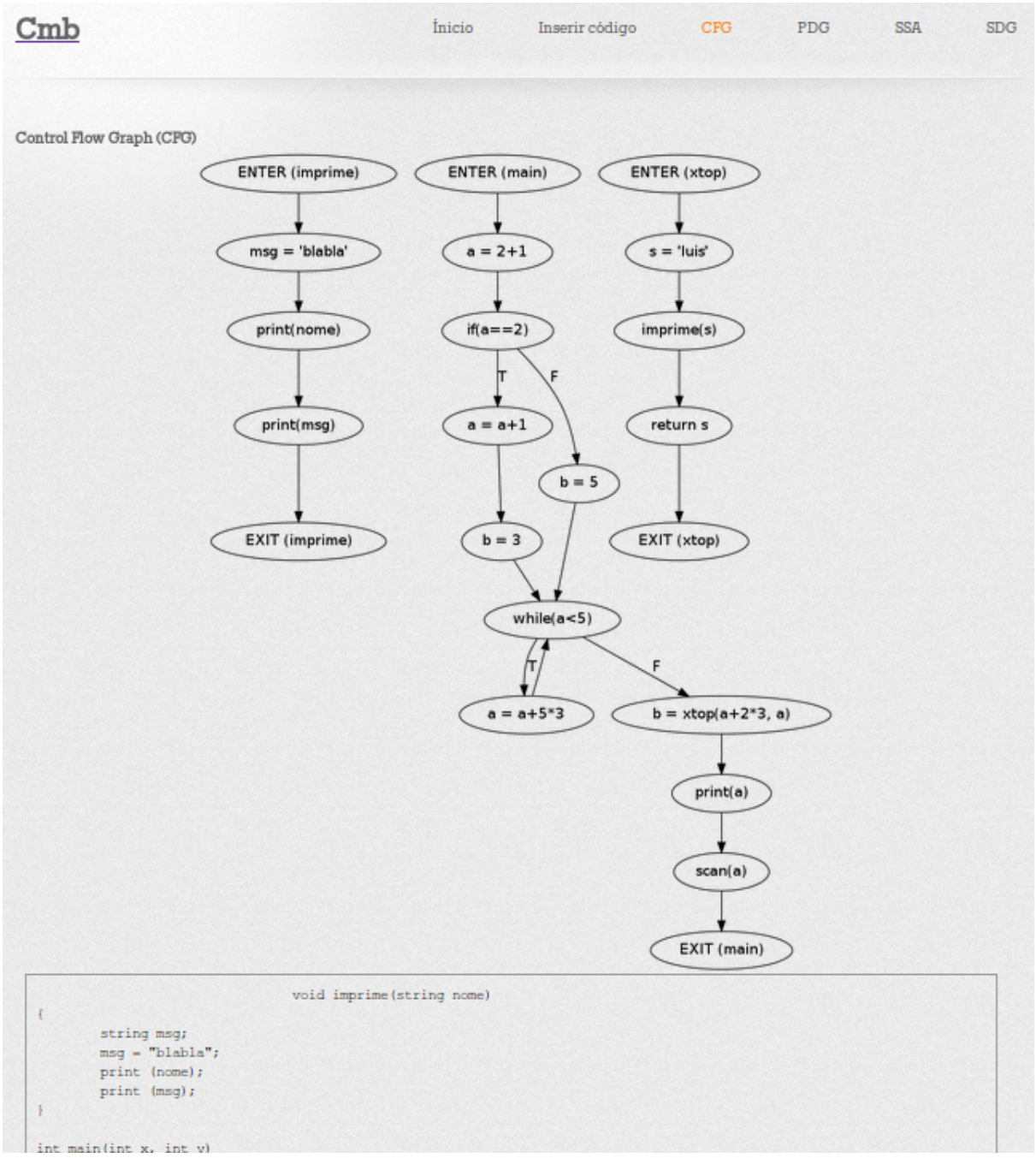

Figura 7: Página do CFG

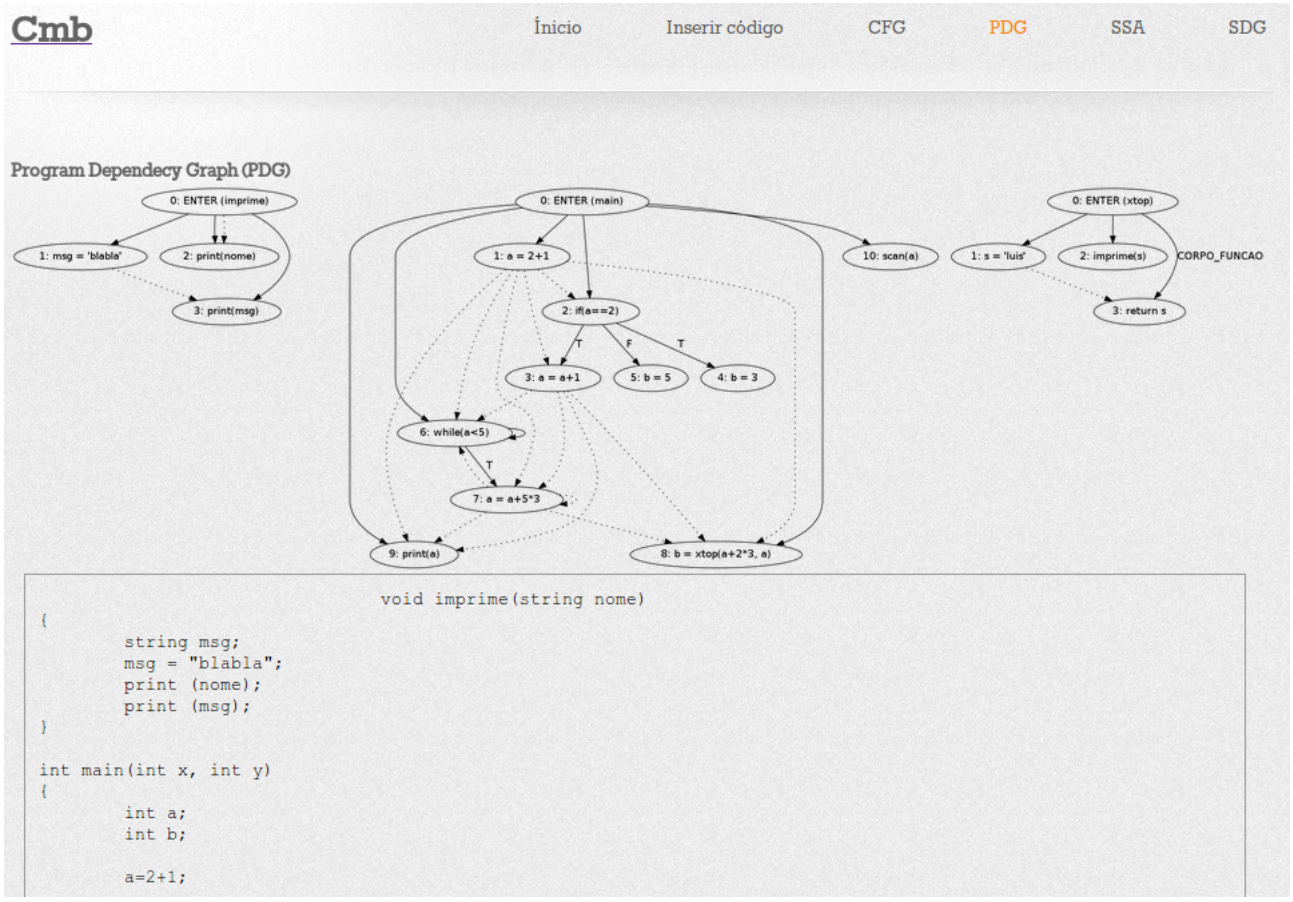

Figura 8: Página do PDG

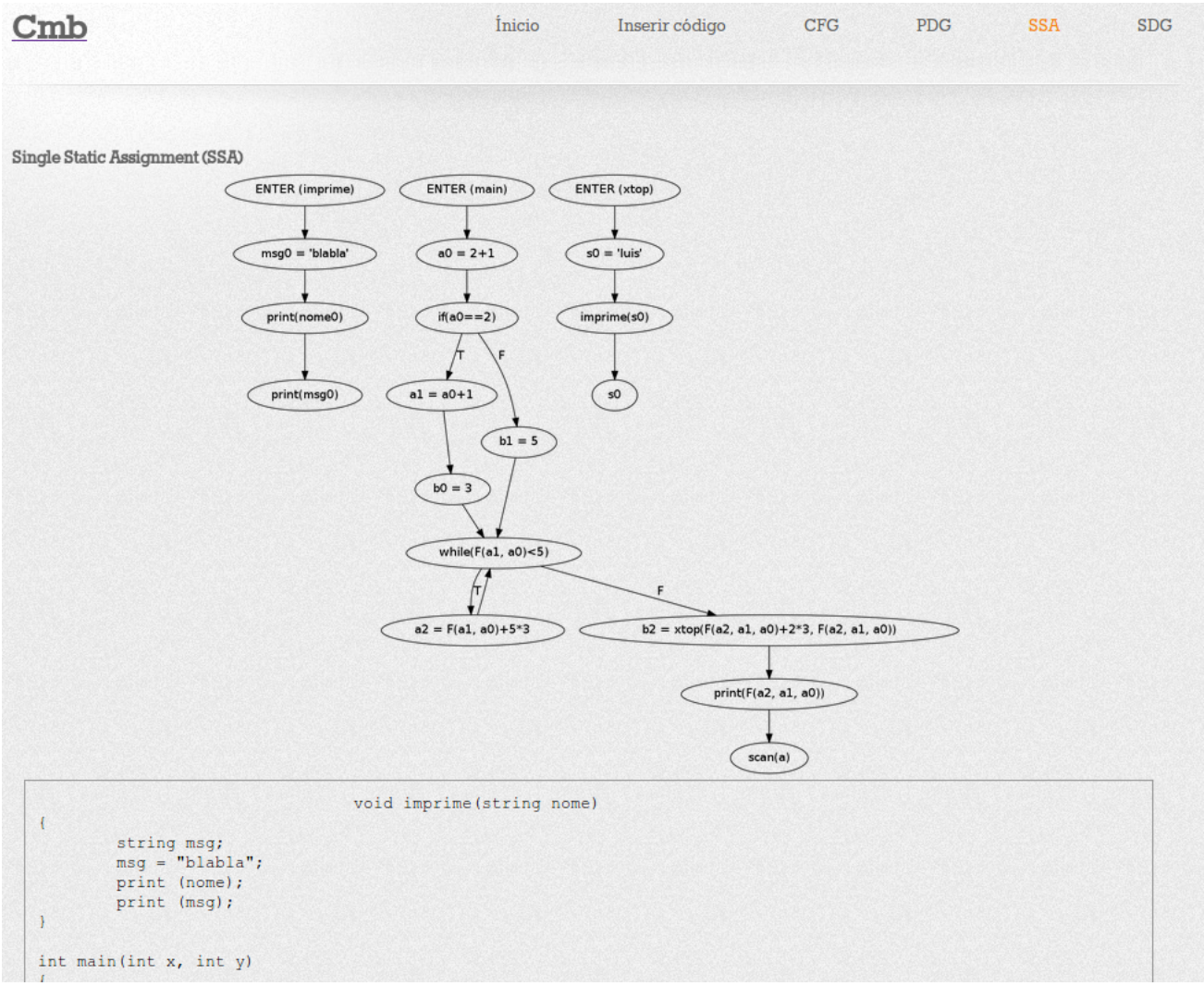

Figura 9: Página do SSA

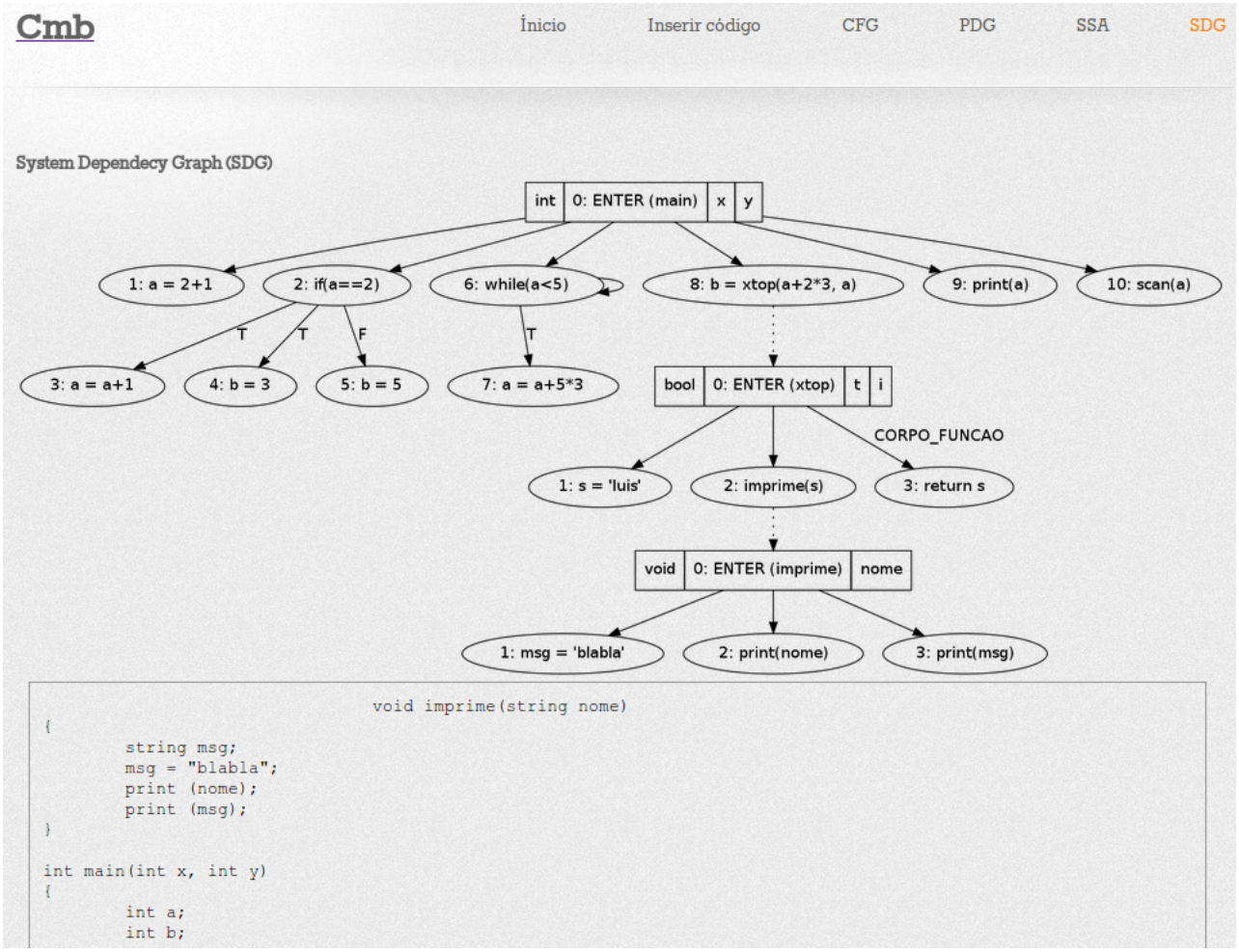

Figura 10: Página do SDG

Estas páginas foram feitas utilizando a linaguagem PHP e são feitas invocações ao java para executar o Run fornecido nas aulas para analisar a gramática e ainda ao dot para gerar os grafos visualmente.

6 Conclusões

Serviu de consolidação da matéria dada no módulo de Análise e Transformação de Software, apesar de haver alguns pormenores que podiam ser melhorados para a resolução dos problemas.