Universidade Federal de Santa Catarina - UFSC
Departamento de Informática e Estatística
Curso de Ciências da Computação
INE5426 - Construção de Compiladores

Luiz João Carvalhaes Motta
Maria Eduarda de Melo Hang
Marina Pereira das Neves Guidolin

Relatório EP3

Análise Semântica e

Gerador de Código Intermediário

Florianópolis, 4 de Dezembro de 2020

1. INTRODUÇÃO

Esse trabalho possui como objetivo implementar um Analisador Semântico e Gerador de Código Intermediário para a linguagem ConvCC-2020-1 conforme proposto no enunciado do Exercício Programa 3.

2. TAREFA ASEM

2.1 CONSTRUÇÃO DA ÁRVORE DE EXPRESSÃO E VERIFICAÇÃO DE TIPOS

Primeiramente, para construção da árvore de expressão, foi feita a separação das produções que derivam expressões aritméticas, chamando-as de produções de gramática EXPA. O arquivo **EXPA.pdf** contém a EXPA, sua SDD e SDT nessa ordem. Preferimos colocá-la separadamente para não poluir o relatório. Acerca da SDD da EXPA, decidimos utilizar a SDD presente na seção 5.3.2 da segunda edição do livro "Compiladores - Princípios, técnicas e ferramentas" para gerar os arrays como uma árvore binária, de forma a facilitar os processos de verificação de tipos e geração de código intermediário.

Para armazenar as árvores de expressões aritméticas foi utilizado uma ArrayList. A ExpressionTree (Árvore de Expressão) é uma árvore binária, sendo que os nodos folhas dessa árvore são operandos e os nodos pais podendo ser tanto operando quanto operadores.

2.1.1 Mostrando que a SDD da EXPA é L-atribuída

O diretório **Grafos/EXPA** contém os grafos de dependência para cada produção da SDD. A figura 1 contém um dos grafos que foram feitos para mostrar que a SDD da EXPA é L-atribuída. Os retângulos arredondados cinza e amarelo representam atributos e código intermediário respectivamente, enquanto que os retângulos brancos são os símbolos da gramática. Considerando esse grafo, percebe-se que todos os atributos

herdados vêm do pai ou do irmão à esquerda, como o caso do atributo **type** do não-terminal **B**, que recebe um atributo herdado do pai **ATRIBSTAT3**, e o atributo **h** do não-terminal **D** que recebe do irmão **B** que está à esquerda de **D**. Da mesma forma, o não-terminal **C** também recebe um atributo herdado do irmão **B**. Além disso, os atributos sintetizados são gerados a partir dos atributos dos filhos e/ou dos próprios atributos do símbolo, como o **expTree** do **ATRIBSTAT3**, cujo valor depende do atributo **sin** do filho **C**. A partir disso, a SDD é considerada L-atribuída, uma vez que contém apenas atributos herdados da esquerda para a direita e sintetizados.

Ao observar todos os grafos presentes nesse diretório, nota-se que todos os atributos herdados e sintetizados seguem essas regras, mostrando que a SDD é L-atribuída.

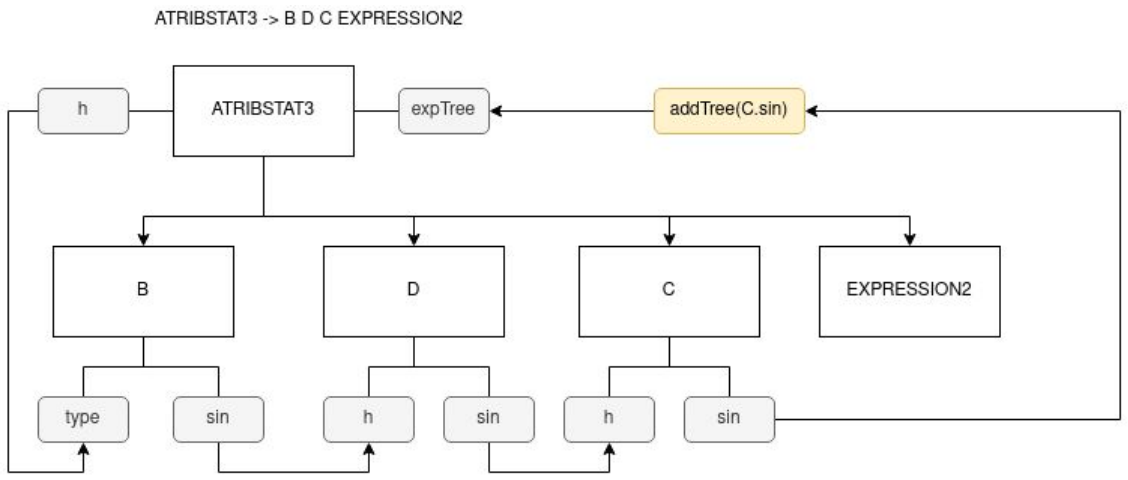

Figura 1: Grafo de dependência para a produção do ATRIBSTAT3

2.1.2 Construção da Árvore

A árvore é criada através da utilização do método **addTree(Node node)** pelas produções de **atribstat2**, **atribstat3** e **numexpression**. Todas estas produções recebem o nodo inicial da expressão aritmética (root) pela sintetização de **c**.

Para passarmos certos atributos aos filhos, utilizamos os colchetes, como pode ser visto na figura 2, que mostra a primeira produção do **atribstat2**, cujo filho **atribstat3** recebe um novo nodo com o valor **IDENT.text** e o tipo do **IDENT**, obtido através do método **getTypeOfIdent**. Para o nodo filho acessar esses parâmetros

recebidos, é necessário colocar todos os parâmetros nas produções do filho entre colchetes, como na figura 3, com o **atribstat3** recebendo um nodo h. Além disso, é possível fazer a cabeça da produção retornar valores com a palavra **returns**, como na figura 4, cuja cabeça **numexpression** retornará um nodo chamado sin. Para acessar esses valores que são retornados, basta usar o operador $<cabeça produção>.<nome atributo retornado>, como é feito na figura 3 na primeira produção de **atribstat3**.

```
atribstat2
    returns [String code, String last]
    locals [ExpressionTree expTree]

    @init {
        $code = "";
        $last = "";
        $expTree = new ExpressionTree();
    }
:

        IDENT atribstat3[new Node($IDENT.text, getTypeOfIdent($IDENT.text))] {$code = $atribstat3.code; $last= $atribstat3.last;}
```

Figura 2: Adição dos nodos à árvore (método addTree)

```
atribstat3[Node h]
    returns [String code, String last]
    locals [ExpressionTree expTree]
    @init {
            $code = "";
            $last = "";
            $expTree = new ExpressionTree();
    }
:
        b[h] d[$b.sin] c[$d.sin] expression2 {$expTree = addTree($c.sin);}

        {
            GenerateReturn generateReturn = generateCode($expTree);
            $code = generateReturn.getCode();
            $last = generateReturn.getLast();
        }
```

Figura 3: Adição dos nodos à árvore (método addTree)

```
numexpression
    returns [Node sin, String code, String last]
    locals [ExpressionTree expTree]
    @init {
        $code = "";
        $last = "";
        $expTree = new ExpressionTree();
    }
:
        term c[$term.sin]
        {
            $sin = $c.sin;
            $expTree = addTree($sin);
            GenerateReturn generateReturn = generateCode($expTree);
        $code = generateReturn.getCode();
        $last = generateReturn.getLast();
    }
;
```

Figura 4: Adição dos nodos à árvore (método addTree)

```
ExpressionTree addTree(Node node) {
    ExpressionTree expTree = new ExpressionTree(node, temporaryCounter);
    Node root = expTree.getRoot();
    String result = expTree.validateTree(root);
    if (result.equals("")) {
        String msg = "The tree below contains invalid expressions\n" + expTree.getRoot().toString();
        notifyErrorListeners("Invalid expression detected");
        throw new ParseCancellationException('\n' + msg);
    }

    trees.add(expTree);
    return expTree;
}
```

Figura 5: Método addTree(Node node)

O método **addTree** recebe como parâmetro um nodo que será utilizado como a raiz da árvore de expressão e retorna essa nova árvore para ser adicionada como atributo da regra que chamá-lo. O método **validateTree**, que é chamado no método **addTree**,

será explicado posteriormente na seção 2.2, uma vez que ele faz parte da validação de expressões. Depois dessa validação, a árvore é adicionada à lista de árvores (**trees**).

Os nodos da árvore são compostos de: valor, filho esquerdo, filho direito e tipo, mas nem todas as propriedades são utilizadas em cada nodo.

```
public class Node {
    private String value;
    private Node left;
    private Node right;
    private String type;
```

Figura 6: Atributos do Nodo

Há três construtores para o Nodo, sendo que cada um deles possui uma utilização.

Criação de um Nodo com valor “array” em que possui o index que está sendo acessado em seu nodo esquerdo (left), o seu valor léxico na direita (right) e o seu tipo.

```
public Node(String value, Node left, Node right, String type) {
    this.value = value;
    this.left = left;
    this.right = right;
    this.type = type;
}
```

Figura 7: Construtor da classe Node com todos os atributos

Criação de um Nodo com o valor de um operador e seus filhos podendo ser tanto operadores quanto operandos.

```
public Node(String value, Node left, Node right) {
    this.value = value;
    this.left = left;
    this.right = right;
    this.type = "";
}
```

Figura 8: Construtor da classe Node sem o parâmetro type

Criação de um Nodo com o valor de um operando e seu respectivo tipo

```
public Node(String value, String type) {
    this.value = value;
    this.left = null;
    this.right = null;
    this.type = type;
}
```

Figura 9: Construtor da classe Node voltado para nodos folha

Após a execução do analisador semântico, um arquivo irá ser gerado no diretório **Out/ExpressionTrees**. Este arquivo possui a árvore T, que foi construída a partir das expressões derivadas das EXPA.

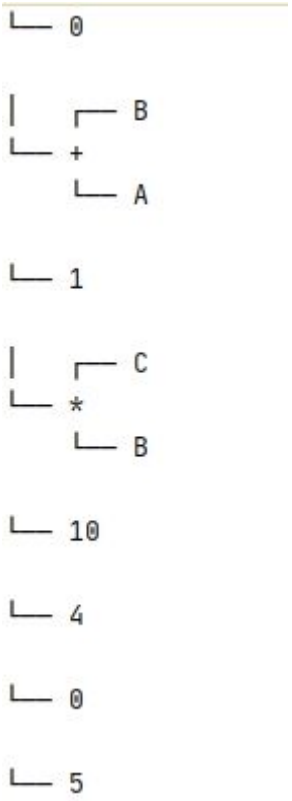

Figura 10: Exemplo da árvore T, gerada após análise semântica do arquivo “example1.ccc”.

2.2 VERIFICAÇÃO DE TIPOS

Após a criação da árvore, ela é validada com o método **validateTree(root)**, que percorre a árvore da expressão de forma recursiva, para então verificar os tipos dos nodos folhas juntamente com o valor do nodo pai.

```
public String validateTree(Node node) {
    if (node.getValue().equals("array")) {
        return node.getType();
    } else {
        if (node.getRight() == null && node.getLeft() == null) {
            return node.getType();
        }
        String left = validateTree(node.getLeft());
        String right = validateTree(node.getRight());
        String operator = node.getValue();
        return validateExpressions(left, right, operator);
    }
}
```

Figura 11: O método validateTree, que valida toda a árvore de expressões

Dependendo do tipo do operador, há uma regra diferente para a realização da verificação.

```
public String validateExpressions(String type1, String type2, String operator) {
    switch (operator) {
        case "+":
            return validateSum(type1, type2);
        case "*":
        case "-":
            return validateSubsMult(type1, type2);
        case "/":
            return validateDivision(type1, type2);
        case "%":
            return validateMod(type1, type2);
    }
    return "";
}
```

Figura 12: O método validateExpressions, que separa cada validação para seu respectivo operador

2.1.2.1 Soma

A soma (+) é permitida apenas entre tipos iguais e entre **float** e **int**.

```
public String validateSum(String type1, String type2) {
    if (type1.equals("int") && type2.equals("int")) {
        return "int";
    } else if (type1.equals("string") && type2.equals("string")) {
        return "string";
    } else if ((type1.equals("float") || type1.equals("int")) && (type2.equals("float") || type2.equals("int"))) {
        return "float";
    } else {
        return "";
    }
}
```

Figura 13: O método validateSum, que valida a soma

Decidimos não permitir a soma de um operando do tipo **string** com um do tipo **int** ou **float**, sendo assim, não é possível ter uma expressão **print** com uma combinação dos dois, pois não adicionamos uma regra de exceção para tal. Para realizar o print de uma **string** seguido de um valor numérico é necessário dois prints como o exemplo a seguir:

```
if (array[n] != 0) {
    print "Resultado: ";
    print array[n];
    return;
}
```

Figura 14: Como printar uma string seguida de um valor numérico

2.1.2.2 Multiplicação, Subtração e Módulo

Não é possível utilizar a multiplicação (*), subtração (-) e módulo (%) para **strings**, e caso ambos sejam do tipo **int** resultará em um **int** caso contrário em um **float**.

```
public String validateSubsMult(String type1, String type2) {
    if (type1.equals("int") && type2.equals("int")) {
        return "int";
    } else if ((type1.equals("float") || type1.equals("int")) && (type2.equals("float") || type2.equals("int"))) {
        return "float";
    } else {
        return "";
    }
}
```

Figura 15: O método validateSubsMult, que valida subtração e multiplicação

2.1.2.3 Divisão

Foi decidido que apenas **int** e **float** podem realizar a divisão (/) e que seu resultado sempre será um **float**.

```
public String validateDivision(String type1, String type2) {
    if ((type1.equals("float") || type1.equals("int")) && (type2.equals("float") || type2.equals("int"))) {
        return "float";
    } else {
        return "";
    }
}
```

Figura 16: O método validateDivision, que valida divisão

2.2. DECLARAÇÃO DE VARIÁVEIS POR ESCOPO E INSERÇÃO DO TIPO NA TABELA DE SÍMBOLOS

O arquivo **DEC.pdf** contém a gramática DEC, sua SDD L-atribuída e SDT. Além disso, o diretório **Grafos/DEC** contém todos os grafos de dependência para a SDD da DEC.

2.2.1 Mostrando que a SDD da DEC é L-atribuída

O diretório **Grafos/DEC** contém os grafos de dependência para cada produção da SDD. A figura 17 contém um dos grafos desenvolvidos para mostrar que a SDD é L-atribuída. Os retângulos contidos nesses grafos têm interpretação análoga aos grafos da EXPA (seção 2.1.1). Ao observar esse grafo, nota-se que o atributo sintetizado **sin** do PARAMLIST depende somente dos atributos dos filhos e não há nenhum atributo

herdado. Como a SDD não apresenta atributos herdados da direita para a esquerda e contém apenas atributos sintetizados, é considerada L-atribuída.

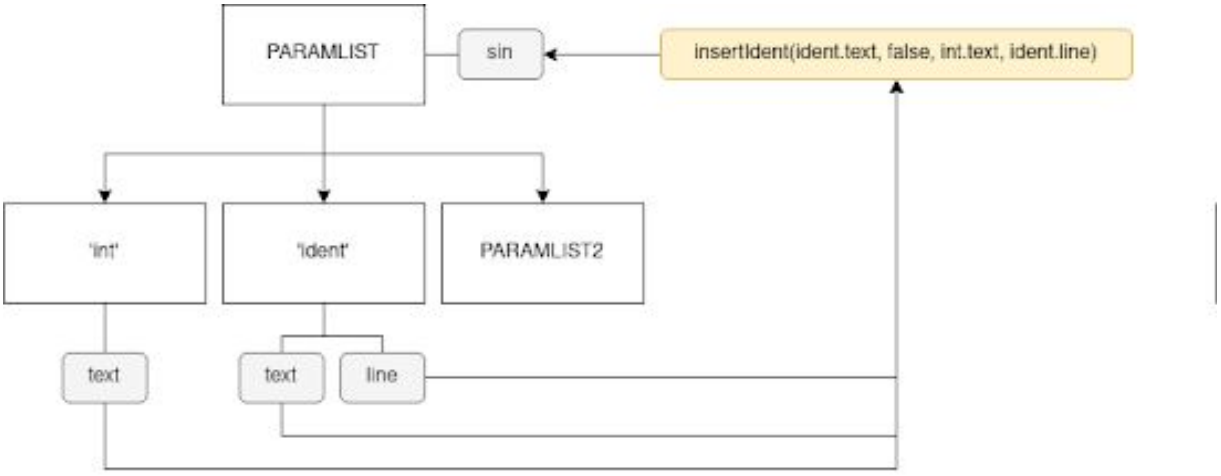

Figura 17: Grafo de dependência para uma produção do PARAMLIST

2.2.2 Solução

Para armazenar os escopos foi utilizado uma pilha ligada. A pilha ligada é uma estrutura de dados que consiste de nodos, sendo que cada nodo aponta para o elemento que está abaixo dele na pilha, que neste caso, é o escopo acima na hierarquia.

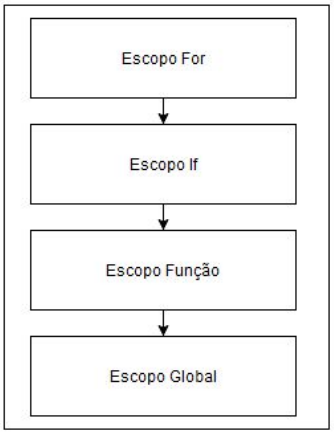

Figura 18: Representação da estrutura de pilha ligada para os nodos

A estrutura do nodo consiste de uma tabela de símbolos, o escopo que o originou e se esse escopo foi criado a partir de um identificador **for**. É utilizado essa variável de controle para verificar a utilização do **break**.

```
public class ScopeNode {
    private SymbolTable symbolTable;
    private ScopeNode previous;
    private boolean isFor;
```

Figura 19: Representação da estrutura da classe ScopeNode

Para adicionar um novo escopo criamos um método **putScope(boolean isFor)** em que cria uma nova tabela de símbolos e informa se o escopo é ou não um **for**. Também criamos um método para remover o escopo após sua utilização **popScope()**

```
void putScope(boolean isFor) {
    scopeStack.push(new SymbolTable(), isFor);
}

void popScope() {
    scopes.add(scopeStack.pop());
}
```

Figura 20: Métodos putScope(booleanIsFor) e popScope()

O primeiro escopo é o global. Ele é adicionado no começo da definição do programa, assim, temos um escopo onde é possível acessar os identificadores globais de funções e variáveis, como é mostrado na figura 21 na parte **@init** das produções do **program**. As anotações **@init** e **@after** são ações semânticas que serão executadas sempre no começo e no fim de qualquer produção de uma cabeça associada. Nesse caso, o **@init** do **program**, irá colocar o escopo global através da ação **putScope(false)**, sendo que o **false** diz que o escopo não é de um for, e inicializa o atributo **code** como uma string vazia. Enquanto que o **@after** irá remover o escopo global ao final de toda a análise, já que é a primeira regra a ser derivada, exportar as árvores de expressão, código intermediário, através do atributo **code** do **program,** e escopos e, por fim,

mostrar mensagens de sucesso em relação à corretude das expressões aritméticas, variáveis declaradas e comandos break. Além disso, a anotação **locals** permite associar atributos que apenas são acessíveis dentro da própria regra, uma vez que não faz sentido a regra **program** retornar um atributo code e também precisamos do atributo **code** na anotação **@after** para exportar o código intermediário resultante.

```
program
    locals [String code]
@init {
    {putScope(false);}
    $code = "";
}
@after {
    popScope();
    Utils.exportExpressionTrees(trees);
    Utils.exportIntermediaryCode($code);
    Utils.exportScopes(scopes);
    Log.success("SUCESS", "Well done! All arithmetic expressions are valid");
    Log.success("SUCESS", "Well done! All declared variables are valid");
    Log.success("SUCESS", "Well done! All break commands are contained within a for statement");
}
:
    statement[false, createLabel("label"), ""]
    {
        if (!$statement.code.isEmpty()) {
            $code = $statement.code ;
        }
    }
    | funclist
    {
        if (!$funclist.code.isEmpty()) {
            $code = $funclist.code;
        }
    }
    |
;
```

Figura 21: Ações semânticas para as produções do PROGRAM

Cada identificador é adicionado juntamente com o seu tipo na tabela de símbolos do escopo atual. O método que insere o identificador na tabela de símbolos, irá fazer uma verificação para checar se o símbolo já existe, e então retorna um “report”, com o resultado da operação. Dependendo do resultado, no caso de já existir o símbolo na tabela, um erro será levantado.

```
void insertIdent(String lexeme, boolean isFunction, String type, int declarationLine) {
    ScopeNode actualScope = scopeStack.peek();
    SemanticReports report = actualScope.getSymbolTable().insert(lexeme, isFunction, type, declarationLine);
    if (report == SemanticReports.IDENT_ALREADY_EXISTS) {
        int previousDeclarationLine = actualScope.getSymbolTable().getEntry(lexeme).getDeclarationLine();
        String msg = "The variable " + lexeme + " has been declared previously at line " + previousDeclarationLine;
        notifyErrorListeners(msg);
        throw new ParseCancellationException('\n' + msg);
    }
}
```

Figura 22: O método insertIdent, que permite inserir um ident no escopo atual

Quando um código é executado, um arquivo é gerado no diretório **Out/Scopes**. Este arquivo utiliza os dados das tabelas de símbolos dos escopos.

```
SCOPE:
Symbol Table:
[A, isFunction = false, type = int, declaration line = 2]
[B, isFunction = false, type = int, declaration line = 2]
[C, isFunction = false, type = int, declaration line = 7]
[x, isFunction = false, type = float, declaration line = 4]
[i, isFunction = false, type = int, declaration line = 9]
[SM, isFunction = false, type = int, declaration line = 3]
[z, isFunction = false, type = float, declaration line = 6]

SCOPE:
Symbol Table:
[R, isFunction = false, type = int, declaration line = 20]
[C, isFunction = false, type = int, declaration line = 16]
[D, isFunction = false, type = int, declaration line = 18]
[y, isFunction = false, type = int, declaration line = 15]
[i, isFunction = false, type = int, declaration line = 19]
[j, isFunction = false, type = int, declaration line = 17]

SCOPE:
Symbol Table:
[principal, isFunction = true, type = function, declaration line = 14]
[func1, isFunction = true, type = function, declaration line = 2]
```

Figura 23: Um exemplo de saída do arquivo “example1.ccc”.

2.3 COMANDOS DENTRO DE ESCOPO

Dependendo do lexema identificado, é feito uma verificação sobre as propriedades do seu escopo, ou uma varredura pelo lexema dentro do tabela de símbolos. Neste caso, temos dois comandos: a verificação do lexema **break**, e da verificação de se um identificador está sendo declarado antes da sua utilização.

2.3.1 Verificação de Break

A verificação da validade do lexema **break** é feita através do boolean **isFor** presente em uma propriedade do nodo responsável por armazenar o escopo. Caso o escopo atual não seja de um **for**, um erro será levantado.

```
void verifyBreak() {
    ScopeNode actualScope = scopeStack.peek();
    if (!actualScope.isFor()) {
        String msg = "A break command was written outside a for statement on line";
        notifyErrorListeners(msg);
        throw new ParseCancellationException('\n' + msg);
    }
}
```

Figura 24: Método que verifica em qual escopo o lexema **break** se encontra

2.3.2 Verificação se foi declarado antes de usar

A verificação da declaração de um identificador antes do seu uso é feita, primeiramente, acessando o escopo atual e realizando uma busca pelo identificador na tabela de símbolos. Caso não seja encontrado, é acessado o escopo acima na hierarquia (o mais próximo do global), e é feita a mesma verificação. Isto se repete até chegar ao escopo global, enquanto o lexema não for encontrado. No final, se o lexema não tiver sido encontrado, um erro é levantado.

```
String getTypeOfIdent(String lexeme) {
    ScopeNode previous = scopeStack.peek();
    while (previous != null) {
         SymbolTableEntry entry = previous.getSymbolTable().getEntry(lexeme);
         if (entry != null) {
            return entry.getType();
         }
         previous = previous.getPrevious();
    }
    String msg = "The identifier " + lexeme + " was used but not defined";
    notifyErrorListeners(msg);
    throw new ParseCancellationException('\n' + msg);
}
```

Figura 25: Método que verifica se um identificador foi declarado antes de ser utilizado

3. TAREFA GCI

3.1 Mostrando que a SDD da GCI é L-atribuída

Para demonstrar o fato de a SDD da GCI é L-atribuída, foram construídos grafos para as produções, eles podem ser encontrados no diretório **Grafos/GCI**. Da mesma forma que os grafos anteriores, os retângulos arredondados cinza representam os atributos, os amarelos representam o código intermediário, já os retângulos brancos são os símbolos da gramática. Tomando como exemplo a produção **IFSTAT1**. Temos que é possível visualizar que os atributos **next**, **breakNext**, **isFor**, pertencentes ao não terminal **STATEMENT**, estão herdando os valores do pai, **IFSTAT1**. Também é possível perceber que os atributos sintetizados são gerados por atributos filhos, conforme a regra **IFSTAT1.code = STATEMENT.code**, que também está representada no grafo da figura 26.

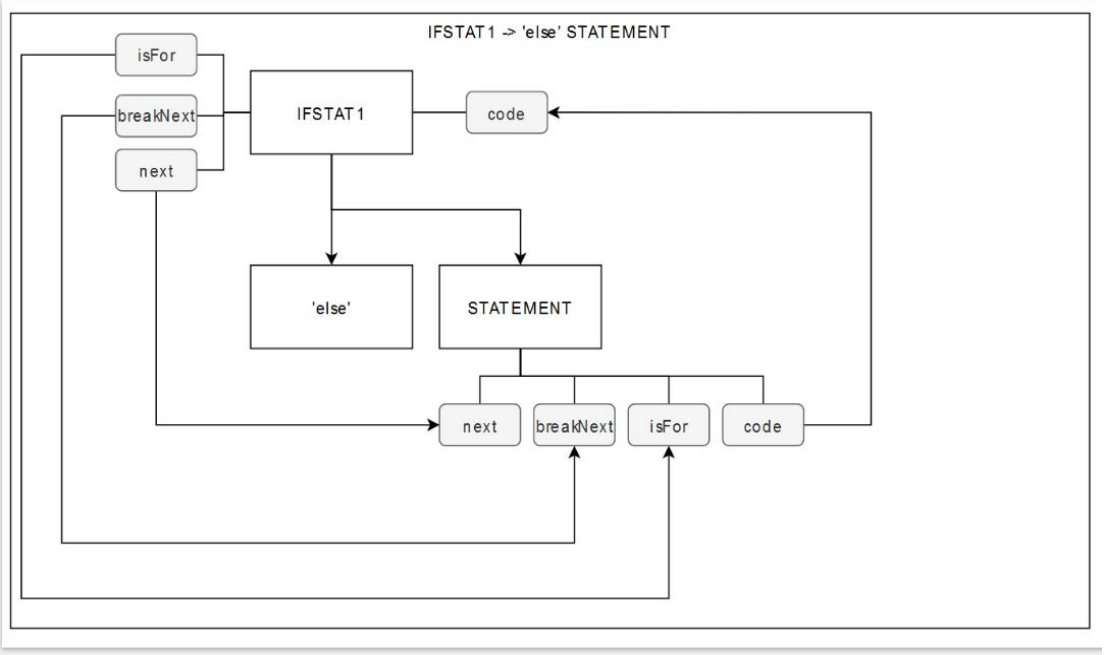

Figura 26: Grafo representando a produção IFSTAT1 → ‘else’ STATEMENT

3.2 Solução

Para a realização da tarefa GCI e aplicar as regras semânticas para a geração de código intermediário para programas na linguagem ConvCC-2020-1 foram criadas duas classes: *Generator.java* e *GenerateReturn.java*. Além disso, o código intermediário é gerado no diretório **Out/CI.**

3.2.1 Classe Generator

A classe *Generator.java* é responsável pela geração de código intermediário. O método **generateThreeAddressCode** está relacionado com a geração de código de três endereços, o **generateCopyCode** ao código de cópia, o **generateIndexedCopyCode** gera as cópias indexadas. Os dois seguintes são responsáveis pela geração dos desvios, sendo o primeiro aos incondicionais (**generateInconditionalDeviationCode**) e o segundo aos condicionais (**generateConditionalDeviationCode**). Por fim, o **generateFunctionCallCode** sintetiza o código intermediário para as chamadas de função, colocando os parâmetros primeiro e depois a chamada com a quantidade de parâmetros.

```java
public class Generator {
	public static String generateThreeAddressCode(String addr1, String addr2, String addr3, String operator) {
		return String.format("%s = %s %s %s\n", addr1, addr2, operator, addr3);
	}

	public static String generateCopyCode(String addr1, String addr2) {
		return String.format("%s = %s\n", addr1, addr2);
	}

	public static String generateIndexedCopyCode(String addr1, String addr2, String index, boolean leftIsIndexed) {
		if (leftIsIndexed) {
			return String.format("%s[%s] = %s\n", addr1, index, addr2);
		} else {
			return String.format("%s = %s[%s]\n", addr1, addr2, index);
		}
	}

	public static String generateInconditionalDeviationCode(String label) {
		return String.format("goto %s\n", label);
	}

	public static String generateConditionalDeviationCode(String expression, String label) {
		return String.format("if %s goto %s\n", expression, label);
	}

	public static String[] generateFunctionCallCode(String functioName, String[] params) {
		StringBuilder sb = new StringBuilder();
		for (String p: params) {
			sb.append("param " + p + '\n');
		}
		String[] result = {sb.toString(), String.format("call %s,%d", functioName, params.length)};
		return result;
	}
}
```

Figura 27: Os métodos da classe generator

A classe **Generator** é chamada pela classe ***ExpressionTree.java,*** dentro do método ***generateCode(Node node, StringBuilder sb)***, conforme a figura 29. O método **generateCode** da classe **ExpressionTree.java** é responsável por gerar o código intermediário para uma árvore de expressão através da recursão. Caso o nodo atual seja um array, ele irá chamar o método **generateCodeArray**, criado especificamente para desenvolver o código intermediário para um árvore com arrays (seu funcionamento será descrito em seguida). Caso seja um nodo de um operador, ele irá percorrer a árvore em pós-ordem, pegando o resultado do filhos, gerando um código de três endereços através do método **generateThreeAddressCode** da classe **Generator**, que será guardado no **StringBuilder**, e retornará a última variável temporária para o nodo pai (o atributo **last** na gramática ConvCC-2020-1).

A figura 28 mostra a ordem ao percorrer uma árvore de expressão sem os arrays, descreveremos abaixo cada passo seguido.

1. Vai para o filho esquerdo
2. Vai para o filho esquerdo novamente
3. Retorna “a”
4. Vai para o filho da direita
5. Retorna “1”
6. Gera o código “t0 = a * 1”
7. Retorna a variável temporária t0
8. Vai para o filho da direita
9. Retorna “b”
10. Gera o código “t1 = t0 + b”
11. Retorna a variável temporária t1

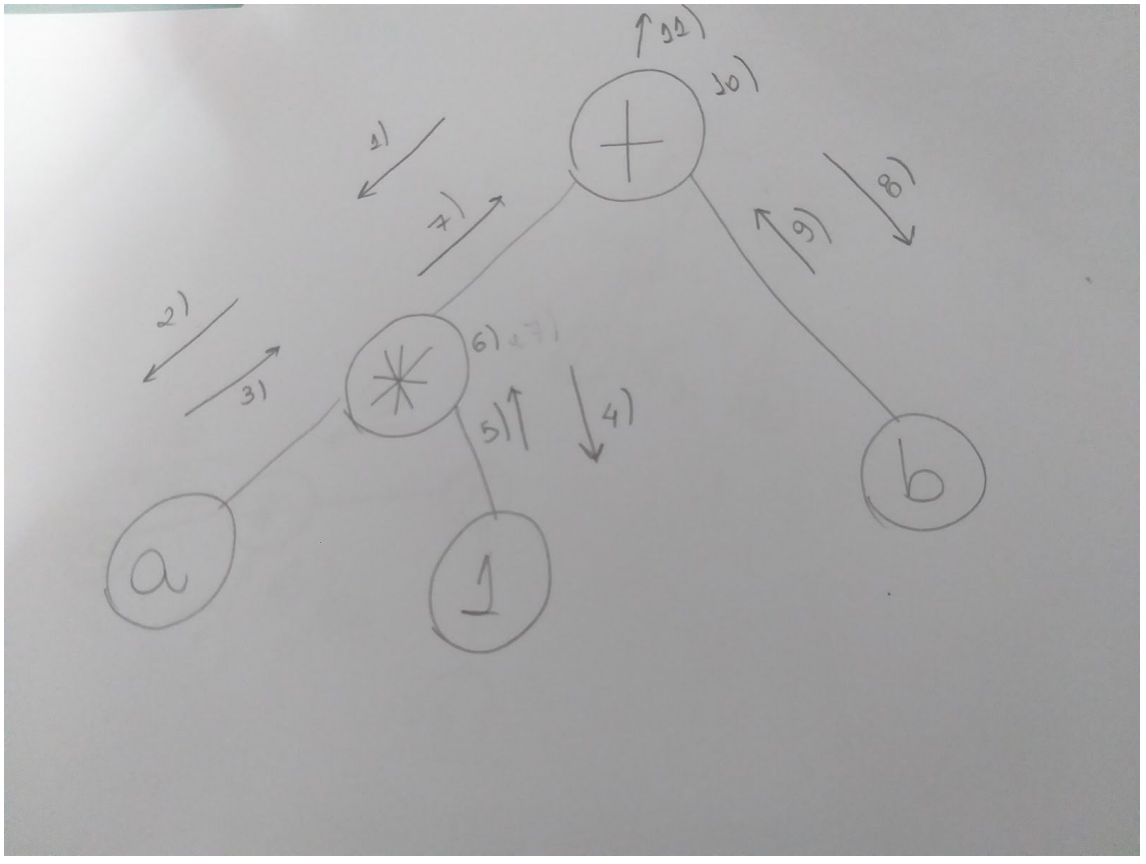

Figura 28: Um exemplo do funcionamento

```
public String generateCode(Node node, StringBuilder sb) {
        if (node.getRight() == null && node.getLeft() == null) {
                return node.getValue();
        } else {
                String temp = "";
                String code = "";
                if (node.getValue().equals("array")) {
                        ArrayList<String> temps = new ArrayList<>();
                        generateCodeArray(node, sb, temps);
                        for (int i = 0; i < temps.size(); i++) {
                                temp = "t" + counter;
                                if (i == 0) {
                                        sb.append(Generator.generateIndexedCopyCode(temp, node.getIdent(), temps.get(i), false));
                                } else {
                                        sb.append(Generator.generateIndexedCopyCode(temp, "t" + (counter-1), temps.get(i), false));
                                }
                                counter++;
                        }
                } else {
                        String left = generateCode(node.getLeft(), sb);
                        String right = generateCode(node.getRight(), sb);
                        temp = "t" + counter;
                        code = Generator.generateThreeAddressCode(temp, left, right, node.getValue());
                }
                sb.append(code);
                counter++;
                return temp;
        }
}
```

Figura 29: Método generateCode(Node node, StringBuilder sb) da classe ExpressionTree.java

A figura 30 mostra o método **generateCodeArray**, que sintetiza o código intermediário para os arrays. O método percorre essa árvore de expressão dos arrays em ordem, pegando a última variável intermediária de cada filho, se houver, e os adicionando à uma ArrayList chamada **temps**. Com a **temps** preenchida, é possível gerar as cópias indexadas de forma a respeitar o acesso dos arrays. A partir dessa lista, o **generateCode** poderá gerar o código intermediário, acessando cada variável temporária na **temps** e gerando o código com o método **generateIndexedCopyCode**. Caso seja a primeira variável temporária, a cópia indexada terá diretamente o **ident** do array como o segundo endereço, a primeira variável como índice e a próxima variável temporária como primeiro endereço, ou seja $\mathbf{t_1 = ident[t_0]}$. Caso contrário, utilizaremos a variável atual do laço como índice, a temporária anterior como segundo endereço e a próxima como primeiro endereço, ou seja $\mathbf{t_{i+1} = t_i[t_{i-1}]}$.

```
public void generateCodeArray(Node node, StringBuilder sb, ArrayList<String> temps) {
        if (node.getRight() != null && node.getLeft() != null) {
                String left = generateCode(node.getLeft(), sb);
                temps.add(left);
                generateCodeArray(node.getRight(), sb, temps);
        }
}
```

Figura 30: O método generateCodeArray

O método da classe **ExpressionTree.java** mencionado anteriormente é utilizado no arquivo ConvCC-20201.g4 (figura 31) para que seja possível percorrer a árvore de expressão a partir do seu nó **root**.

Além disso, a classe **GenerateReturn** foi desenvolvida devido à necessidade de ter tanto o código como a última variável temporária da árvore sendo retornados para as produções da gramática, a figura 33 mostra a classe e seus atributos.

```
GenerateReturn generateCode(ExpressionTree expTree) {
    StringBuilder sb = new StringBuilder();
    Node root = expTree.getRoot();
    GenerateReturn generateReturn = new GenerateReturn(expTree.generateCode(root, sb), sb.toString());
    temporaryCounter = expTree.getCounter();
    return generateReturn;
}
```

Figura 31: Método generateCode(ExpressionTree expTree) do arquivo ConvCC-20201.g4

Pode-se observar também que o **generateCode(ExpressionTree expTree)** também é utilizado nas produções, para que, a partir da SDT, seja possível a geração de código intermediário, conforme o exemplo da figura 32.

```
lvalue[String h]
    returns [Node node, String code, String last]
    locals [ExpressionTree expTree]
:
        IDENT b[new Node(h + $IDENT.text, getTypeOfIdent($IDENT.text))] {$node = $b.sin; $expTree = addTree($node);}
        {
       GenerateReturn generateReturn = generateCode($expTree);
       $code = generateReturn.getCode();
       $last = generateReturn.getLast();
        }
;
```

Figura 32: Produções de LVALUE utilizando o método generateCode(ExpressionTree expTree)

```java
public class GenerateReturn {
	private String last;
	private String code;

	public GenerateReturn(String last, String code) {
		this.last = last;
		this.code = code;
	}

	public String getLast() {
		return last;
	}

	public String getCode() {
		return code;
	}
}
```

Figura 33: A classe GenerateReturn, seus atributos e métodos