UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

CURSO DE BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

INE5426 – Construção de Compiladores

Trabalho 2 -
Análise Semântica e Gerador de Código Intermediário

Project Group 01

Bruno George Moraes - 14100825

Marcelo José Dias - 15205398

Renan Pinho Assi - 12200656

Vinícius Schwinden Berkenbrock -16100751

.

FLORIANÓPOLIS

2019/2

SUMÁRIO

1. INTRODUÇÃO

Neste trabalho realizamos duas tarefas, Análise Semântica e Geração de Código Intermediário, também nomeadas aqui como ASem e GCI respectivamente, em cima da gramática dada CCC-2019-2 no formato BNF, vista abaixo.

A gramática possui a inclusão de algumas produções e poucas alterações da gramática já realizada em nosso Trabalho TP1 da disciplina, então essas novas produções ou alteradas foram transformadas para Normal, removemos a recursão à esquerda e a fatoramos para incluir ao restante das produção em nosso código. Todos os passos novamente de transformação, retirada de recursão e fatoração, serão emitidos em detalhes pois já foi parte do trabalho anterior, mas segue um exemplo de uma das novas produções:

PARAMLIST → **((** ***int*** | ***float*** | ***string*** **)** ***ident*****,** *PARAMLIST* | **(** ***int*** | ***float*** | ***string*** **)** ***ident*****)?**

Produção PARAMLIST no formato BNF.

Transformamos para o formato padrão, analisando a expressão. O primeiro trecho (int|float|string) precisa ser distribuído com o restante "ident, PARAMLIST".

(***int*** | ***float*** | ***string*** **)** ***ident*****,** *PARAMLIST* → ***int ident*****,** *PARAMLIST* |
float ident**,** *PARAMLIST* |
string ident**,** *PARAMLIST*

O segundo trecho (ie a segunda produção) **(** ***int*** | ***float*** | ***string*** **)** ***ident*** segue o mesmo passo e forma: ***int ident*** | ***float ident*** | ***string ident***

Toda a produção está em torno de um ?, o que significa que pode ocorrer uma vez ou nenhuma. A interpretação final da produção é a seguinte:

PARAMLIST → (***int ident,*** *PARAMLIST* | ***float ident,*** *PARAMLIST* |
string ident, *PARAMLIST*) | (***int ident*** | ***float ident*** | ***string ident***) | ϵ

O símbolo ϵ é incluído como opção de derivação devido ao ? da notação de expressões regulares.

Neste ponto já é perceptível a necessidade de uma fatoração, pois temos uma repetição de símbolos no ínicio de produções diferentes, como em:

"*PARAMLIST* → ***int ident*** , *PARAMLIST*" e "*PARAMLIST* → ***int ident*** "

Então criamos um não-terminal PARAMLIST2 para receber as variações da produção e deixamos o que se repete só uma vez.

Versão final na Forma Normal de Chomsky:

PARAMLIST → ***int ident*** *PARAMLIST2*
PARAMLIST → ***float ident*** *PARAMLIST2*
PARAMLIST → ***string ident*** *PARAMLIST2*
PARAMLIST → ϵ
PARAMLIST2 → **,** *PARAMLIST*
PARAMLIST2 → ϵ

O mesmo procedimento é aplicado a todas as produções que estão diferentes do trabalho anterior TP1 ou com as novas produções e o resultado é utilizado nas tarefas a seguir e em nosso código.

PROGRAM → (*STATEMENT* | *FUNCLIST*)?
FUNCLIST → *FUNCDEF FUNCLIST* | *FUNCDEF*
FUNCDEF → *def ident* (*PARAMLIST*){ *STATELIST* }
PARAMLIST → ((*int* | *float* | *string*) *ident*, *PARAMLIST* | (int | float | string) ident)?
STATEMENT → (*VARDECL*; | *ATRIBSTAT*; | *PRINTSTAT*; | *READSTAT*; | *RETURNSTAT*; | *IFSTAT* | *FORSTAT* | { *STATELIST* } | *break* ; | ;)
VARDECL → (*int* | *float* | *string*) *ident* ([*int constant*])*
ATRIBSTAT → *LVALUE* = (*EXPRESSION* | *ALLOCEXPRESSION* | *FUNCCALL*)
FUNCCALL → *ident* (*PARAMLISTCALL*)
PARAMLISTCALL → (*ident*, *PARAMLISTCALL* | *ident*)?
PRINTSTAT → *print EXPRESSION*
READSTAT → *read LVALUE*
RETURNSTAT → *return*
IFSTAT → *if*(*EXPRESSION*) *STATEMENT* (*else STATEMENT*)?
FORSTAT → *for*(*ATRIBSTAT*; *EXPRESSION*; *ATRIBSTAT*) *STATEMENT*
STATELIST → *STATEMENT* (*STATELIST*)?
ALLOCEXPRESSION → *new* (*int* | *float* | *string*) ([*NUMEXPRESSION*])+
EXPRESSION → *NUMEXPRESSION* ((< | > | <= | >= | == | !=) *NUMEXPRESSION*)? *NUMEXPRESSION* → *TERM* ((+ | −) *TERM*)*
TERM → *UNARYEXPR* ((* | \ | %) *UNARYEXPR*)*
UNARYEXPR → ((+ | −))? *FACTOR*
FACTOR → (*int_constant* | *float_constant* | *string_constant* | *null* | *LVALUE* | (*NUMEXPRESSION*))
LVALUE → *ident*([*NUMEXPRESSION*])*

Gramática CCC-2019-2 em sua forma original BNF.

2. TAREFA ASem

Esse capítulo apresenta o desenvolvimento da parte de Análise Semântica, dividido em 5 passos:

- Construção de uma árvore de sintaxe para produções que geram expressões aritméticas
- Inserção do tipo de variáveis na tabela de símbolos
- Verificação de tipos em expressões aritméticas
- Declaração de variáveis por escopo, e
- Verificação do comando break em um escopo de comando de repetição.

2.1 CONSTRUÇÃO DA ÁRVORE DE SINTAXE

2.1.1 Gramática EXPA

Separamos as produções que geram expressões aritméticas presentes na CCC-2019-2, iniciando-se no não terminal *NUMEXPRESSION*.

NUMEXPRESSION → *TERM* ((+ |−) *TERM*)*
TERM → *UNARYEXPR* ((* | \ | %) *UNARYEXPR*)*
UNARYEXPR → ((+ |−))? *FACTOR*
FACTOR → (int_constant | float_constant | string_constant | null |
LVALUE | (*NUMEXPRESSION*))
LVALUE → ident([*NUMEXPRESSION*])*

Gramática EXPA - forma BNF

2.1.2 SDD L-Atribuída:

Construção de uma SDD, contendo as produções já na forma normal de Chomsky, fatorada e sem recursão à esquerda.

Criamos as regras semânticas com o intuito da criação da árvore de sintaxe, então nas produções que contém operadores aritméticos como em *NUM2* e *TERM2*, temos as regras de criar nodos na árvore no formato Node(<operador>, <operando a esquerda>, <operando a direita>), sendo que os operandos podem ser um valor herdado ou sintetizado de outro nodo.

O sinais ‘+’ e ‘-’ em *FACTOR,* representam o sinal de um elemento e não operadores.

Produções da EXPA	Regras Semânticas
NUMEXPRESSION →TERM NUM2	NUMEXPRESSION.node = NUM2.syn NUM2.inh = TERM.node
NUM2 → + TERM NUM2’	NUM2’.inh = new Node (+, NUM2.inh, TERM.node) NUM2.syn = NUM2’.syn
NUM2 → - TERM NUM2’	NUM2’.inh = new Node (-, NUM2.inh, TERM.node) NUM2.syn = NUM2’.syn
NUM2 → ϵ	NUM2.syn = NUM2.inh
TERM → UNARYEXPR TERM2	TERM.node = TERM2.syn TERM2.inh = UNARYEXPR.node
TERM2 → * UNARYEXPR TERM2’	TERM2’.inh = new Node (*, TERM2.inh, UNARYEXPR.node) TERM2.syn = TERM2’.syn
TERM2 → \ UNARYEXPR TERM2’	TERM2’.inh = new Node (\, TERM2.inh, UNARYEXPR.node) TERM2.syn = TERM2’.syn
TERM2 → % UNARYEXPR TERM2’	TERM2’.inh = new Node (%, TERM2.inh, UNARYEXPR.node) TERM2.syn = TERM2’.syn
TERM2 → ϵ	TERM2.syn = TERM2.inh
UNARYEXPR → + FACTOR	UNARYEXPR.syn = '+' FACTOR.syn
UNARYEXPR → - FACTOR	UNARYEXPR.syn = '-' FACTOR.syn
UNARYEXPR → FACTOR	UNARYEXPR.node = FACTOR.node
FACTOR → **int_constant**	FACTOR.node = new Leaf (int_constant, int_constant.val)
FACTOR → **float_constant**	FACTOR.node = new Leaf (float_constant, float_constant.val)
FACTOR → **string_constant**	FACTOR.node = new Leaf (string_constant, string_constant.val)
FACTOR → null	FACTOR.node = new Leaf (null, null.val)
FACTOR → LVALUE	FACTOR.node = LVALUE.node

FACTOR → (NUMEXPRESSION)	FACTOR.node = NUMEXPRESSION.node
LVALUE → ident ALLOC3	LVALUE.syn = ident.lexval + ALLOC3.val
ALLOC3 → [NUMEXPRESSION] ALLOC3`	ALLOC3.syn = ALLOC3'.syn
ALLOC3 → ϵ	ALLOC3.syn = ALLOC3.inh

Obs1.: O sinal de + na regra semântica de LVALUE significa a concatenação dos elementos.

2.1.3 Mostrando que a SDD EXPA realmente é L-atribuída

Para mostrar que a SDD é do tipo L-atribuída, verificamos as produções que possuem atributos herdados, os quais devem ter origem do elemento pai ou de um irmão à esquerda. Pois pela definição será uma L-atribuída se somente possuir atributos sintetizados, ou em caso de possuir atributos herdados, estes virão dos pais ou irmão à esquerda (AHO; SETHI; ULLMAN, 1986).

Sabemos que há atributos herdados, analisaremos cada um deles.

NUMEXPRESSION →TERM NUM2	NUMEXPRESSION.node = NUM2.syn NUM2.inh = TERM.node

Em { NUM2.inh = TERM.node } o atributo herdado de NUM2 está vindo de TERM, que é seu irmão à esquerda. Condição válida.

NUM2 → + TERM NUM2'	NUM2'.inh = new Node (+, NUM2.inh, TERM.node) NUM2.syn = NUM2'.syn

Em {NUM2'.inh = new Node (+, NUM2.inh, TERM.node)} o atributo herdado de NUM2' utiliza uma função para criação de um nodo e utiliza NUM2.inh que é seu pai, e TERM.node que é seu irmão à esquerda. Condição válida.

NUM2 → - TERM NUM2'	NUM2'.inh = new Node (-, NUM2.inh, TERM.node)

	NUM2.syn = NUM2'.syn

Aqui se repete a situação acima, NUM2.inh herda atributos do irmão à esquerda e do pai. Condição válida.

TERM → UNARYEXPR TERM2	TERM.node = TERM2.syn TERM2.inh = UNARYEXPR.node

Em {TERM2.inh = UNARYEXPR.node}, TERM2.inh herda do irmão à esquerda UNARYEXPR.node. Condição válida.

TERM2 → * UNARYEXPR TERM2'	TERM2'.inh = new Node (*, TERM2.inh, UNARYEXPR.node) TERM2.syn = TERM2'.syn
TERM2 → \ UNARYEXPR TERM2'	TERM2'.inh = new Node (\, TERM2.inh, UNARYEXPR.node) TERM2.syn = TERM2'.syn
TERM2 → % UNARYEXPR TERM2'	TERM2'.inh = new Node (%, TERM2.inh, UNARYEXPR.node) TERM2.syn = TERM2'.syn

Para os 3 casos acima, *TERM2'*.inh cria um nodo herdando do pai *TERM2*, e do irmão à esquerda *UNARYEXPR* na função new Node(<operador>, *TERM2*.inh, *UNARYEXPR*.nod). Condição válida.

Todos os atributos herdados foram validados pela definição, o restante são atributos sintetizados, comprovando que nossa SDD é do tipo L-atribuída.

2.1.4 SDT EXPA

A seguir o esquema de tradução, onde colocamos as ações já definidas na SDD, junto com as produções gramaticais. Ações na qual atributos herdados recebem dados são postos logo a esquerda do elemento em questão. Como em {*NUM.inh = TERM.node*} se posicionando antes de NUM logo na primeira produção de EXPA.

Ações com atributos sintetizados se posicionam ao final da produção.

SDT para a SDD de EXPA

NUMEXPRESSION → TERM {NUM2.inh = TERM.node} NUM2 {NUMEXPRESSION.node = NUM2.syn }

NUM2 → + TERM {NUM2'.inh = new Node (+, NUM2.inh, TERM.node)} NUM2' {NUM2.syn = NUM2'.syn}

NUM2 → - TERM {NUM2'.inh = new Node (-, NUM2.inh, TERM.node)} NUM2' {NUM2.syn = NUM2'.syn}

NUM2 → ϵ {NUM2.syn = NUM2.inh}

TERM → UNARYEXPR {TERM2.inh = UNARYEXPR.node} TERM2 {TERM.node = TERM2.syn }

TERM2 → ∗ UNARYEXPR {TERM2'.inh = new Node (*, TERM2.inh, UNARYEXPR.node) } TERM2' {TERM2.syn = TERM2'.syn}

TERM2 → \ UNARYEXPR {TERM2'.inh = new Node (\, TERM2.inh, UNARYEXPR.node) } TERM2' {TERM2.syn = TERM2'.syn}

TERM2 → % UNARYEXPR {TERM2'.inh = new Node (%, TERM2.inh, UNARYEXPR.node) } TERM2' {TERM2.syn = TERM2'.syn}

TERM2 → ϵ {TERM2.syn = TERM2.inh}

UNARYEXPR → + FACTOR {UNARYEXPR.syn = '+' FACTOR.syn}

UNARYEXPR → - FACTOR {UNARYEXPR.syn = '-' FACTOR.syn}

UNARYEXPR → FACTOR {UNARYEXPR.syn = FACTOR.syn}

FACTOR → int_constant {FACTOR.node = new Leaf (int_constant, int_constant.val)}

FACTOR → float_constant {FACTOR.node = new Leaf (float_constant, float_constant.val)}

FACTOR → string_constant {FACTOR.node = new Leaf (string_constant, string_constant.val)}

FACTOR → null {FACTOR.node = new Leaf (null, null.val)}

FACTOR → LVALUE {FACTOR.node = LVALUE.node}

FACTOR → (NUMEXPRESSION) {FACTOR.node = NUMEXPRESSION.node}

LVALUE → ident ALLOC3 {LVALUE.syn = ident.lexval + ALLOC3.val }

ALLOC3 → [NUMEXPRESSION] ALLOC3` {ALLOC3.syn = ALLOC3'.syn}

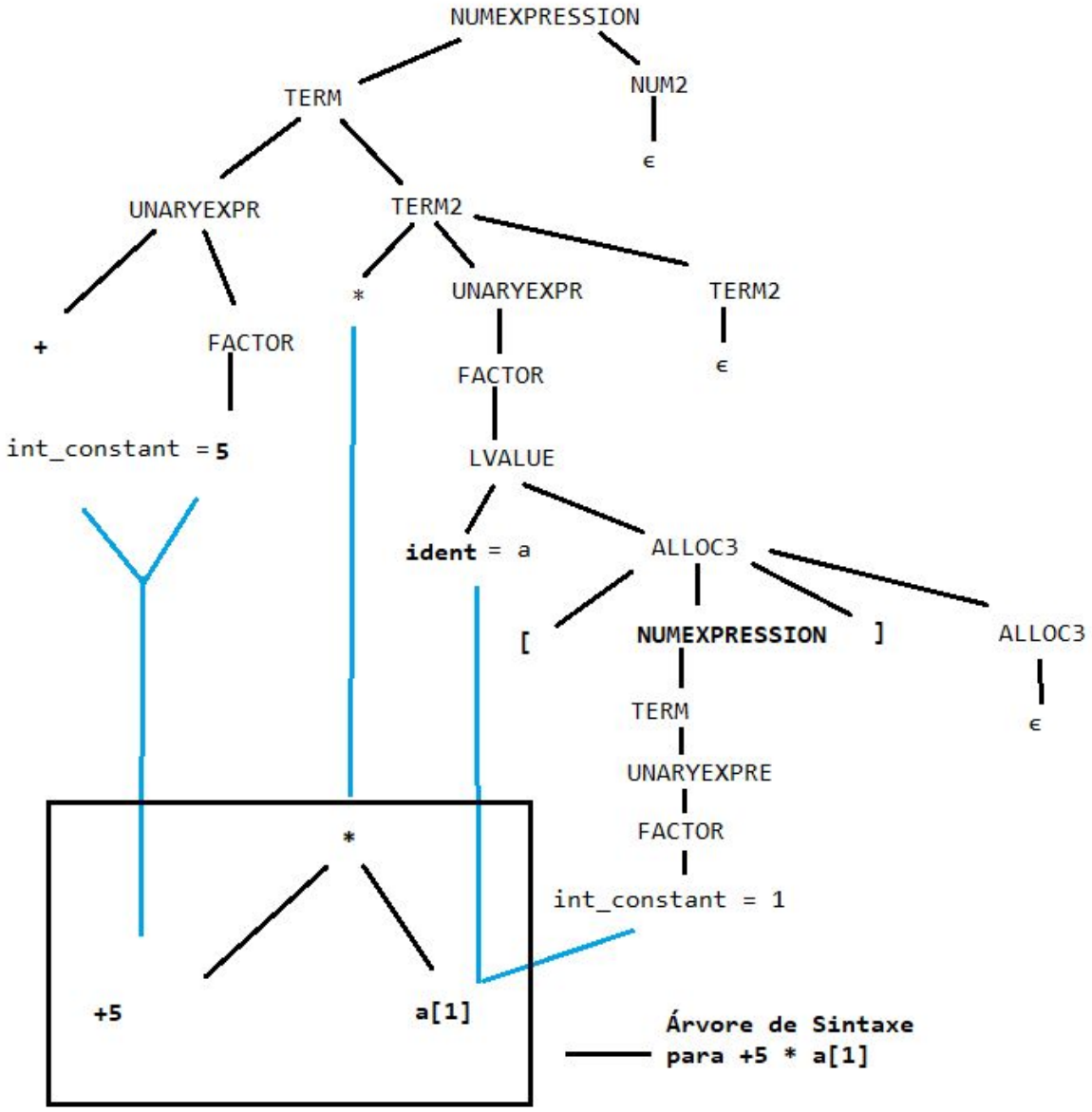
ALLOC3 → ϵ {ALLOC3.syn = ALLOC3.inh}

2.1.5 Árvore de Sintaxe para EXPA

Geramos um árvore de sintaxe a partir de uma árvore gramatical.

Exemplo de uma derivação para entrada ***+ 5 * a[1]*** na gramática parcial denominada de EXPA.

A figura a seguir mostra um árvore gramatical no topo, com suas derivações mostradas com traços pretos, e os traços em azul mostram o operador * tornando raiz na árvore de sintaxe abaixo, e os operandos virando folha, o + representa o sinal do número 5 e não um operador, eles são concatenados para formar a folha +5, o mesmo ocorre com a representação do array a[1] que se torna outra folha na parte direita.



Na imagem a seguir derivamos uma expressão um pouco maior, **(7 + 7) \ 7** e a partir da árvore gramatical derivada geramos a árvore sintática na parte inferior, com setas vermelhas para ligar operandos e operadores.

Setas pontilhadas pretas representam as derivações da produções para a árvore gramatical.

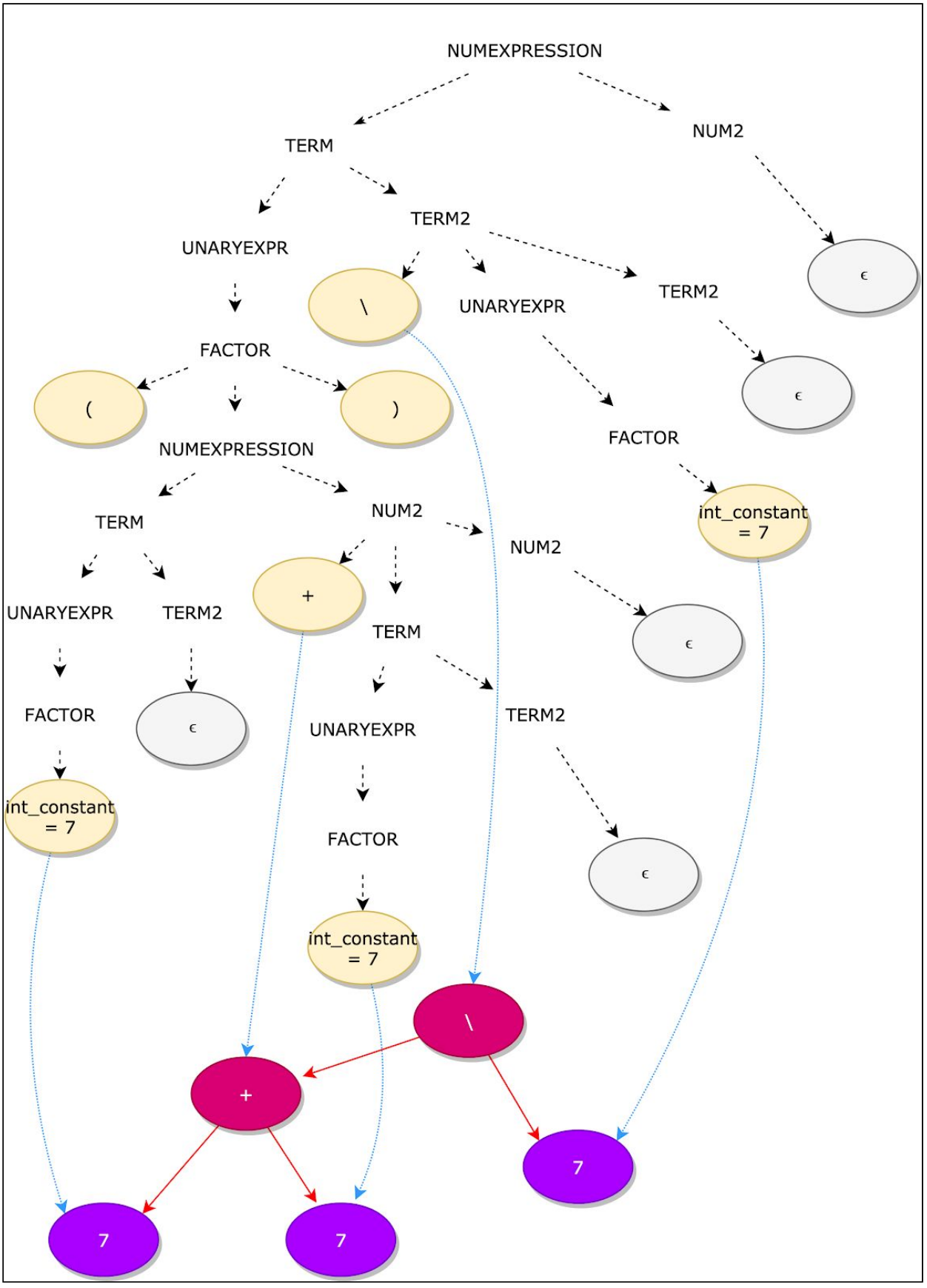


Figura Geração de Árvore de Sintaxe para (7 + 7) \ 7

2.2 INSERÇÃO DO TIPO NA TABELA DE SÍMBOLOS

Aqui realizamos a análise semântica para inserção de tipos, criando uma SDD e uma SDT com as regras semânticas, e quatro passos são realizados:

- Separação de produções da gramática que derivam declarações de variável
- Criação de uma SDD L-atribuída para inserção de tipos
- Verificação de que a SDD criada realmente é uma L-atribuída
- Criação de uma SDT a partir da SDD anterior.

2.2.1 Gramática DEC

Separamos da gramática CCC-2019-2 as produções que derivam declarações de variáveis e a chamaremos de gramática DEC.

2.2.2 SDD L-atribuída

Produções DEC	Regras Semânticas	
VARDECL → string ident VAR2	ident.type.inh = string addType(ident.entry , type= ident.type.inh, array = VAR2.syn)	ident.type.inh = string
VARDECL → float ident VAR2	ident.type.inh = float addType(ident.entry , type= ident.type.inh, array = VAR2.syn)	ident.type.inh = float
VARDECL → int ident VAR2	ident.type.inh = int addType(ident.entry , type= ident.type.inh, array = VAR2.syn)	ident.type.inh = int
VAR2 → [int_constant] VAR2'	VAR2’.inh = VAR2.inh VAR2.syn = VAR2'.syn	VAR2.syn = VAR2'.syn

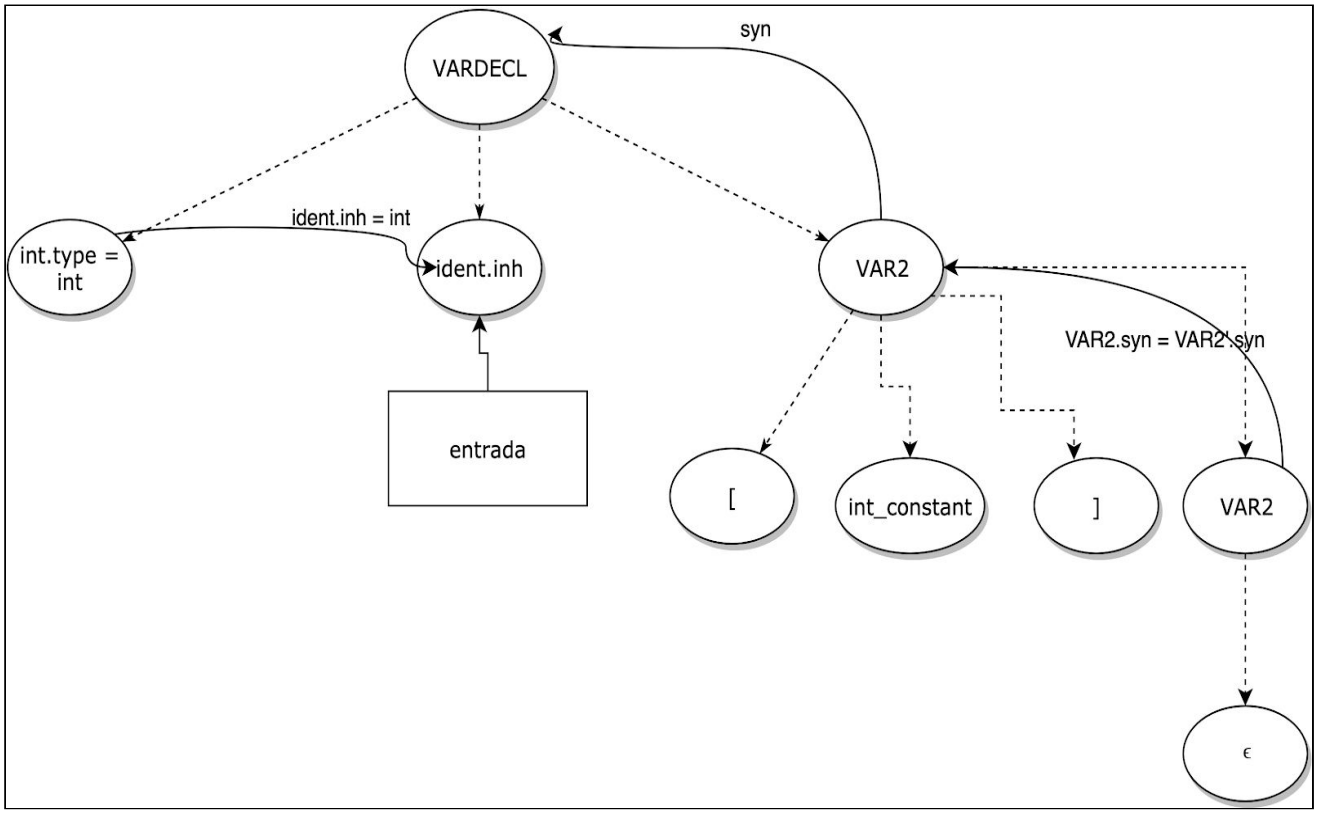
VAR2 → ϵ	VAR2.syn = VAR2.inh	
PARAMLIST -> string ident PARAMLIST2	addType(ident.entry , type= ident.type.inh, ARRAY = null)	ident.type.inh = string
PARAMLIST -> float ident PARAMLIST2	addType(ident.entry , type= ident.type.inh, ARRAY = null)	ident.type.inh = float
PARAMLIST -> int ident PARAMLIST2	addType(ident.entry , type= ident.type.inh, ARRAY = null)	ident.type.inh = int
PARAMLIST -> &	PARAMLIST .syn = PARAMLIST.inh	
PARAMLIST2 -> , PARAMLIST	PARAMLIST.inh = PARAMLIST2.inh	PARAMLIST2.syn = PARAMLIST.syn
PARAMLIST2 -> &	PARAMLIST2.syn = PARAMLIST2.inh	
ALLOCEXPRESSION -> new ALLOC2	ALLOCEXPRESSION.type= ALLOC2.type ALLOCEXPRESSION.array = ALLOC2.array	
ALLOC2 -> string [NUMEXPRESSION] ALLOC3	ALLOC2.type= string, ARRAY = ALLOC3.syn)	
ALLOC2 -> float [NUMEXPRESSION] ALLOC3	ALLOC2.type= float, ARRAY = ALLOC3.syn)	
ALLOC2 -> int [NUMEXPRESSION] ALLOC3	ALLOC2.type= int, ARRAY = ALLOC3.syn)	
ALLOC3 -> [NUMEXPRESSION] ALLOC3`		ALLOC3.syn = ALLOC3'.syn
ALLOC3 -> e	ALLOC3.syn = ALLOC.inh	

2.2.3 Mostrando que é uma SDD L-atribuida

Nas 3 primeiras produções o terminal ***ident*** herda o atributo do irmão à esquerda, uma herança da esquerda para direita, sendo herdado os tipos (string, float e int) respectivamente.

Todas os outros atributos são sintetizados, portanto obedecendo a regra da L-atribuída, que caso haja atributos herdados, eles só podem ser de produções irmãs à esquerda e/ou dos nodos pai.

No grafo a seguir mostramos visualmente o tipo da variável sendo herdado do nodo irmão à esquerda, o atributo de ident recebe o tipo e os outros nodos apenas sintetizam. Garantindo o tipo da variável e de que nossa SDD é L-atribuída.



Grafo de dependências para VARDECL

2.2.4 Criação de uma SDT para DEC

SDT
VARDECL → string {ident.type.inh = string} ident VAR2 {addType(ident.entry , type= ident.type.inh, array = VAR2.syn)}
VARDECL → float {ident.type.inh = float} ident VAR2 {addType(ident.entry , type= ident.type.inh, array = VAR2.syn)}

VARDECL → int {ident.type.inh = int} ident VAR2 {addType(ident.entry , type= ident.type.inh, array = VAR2.syn)}
VAR2 → [int_constant] {VAR2'.inh = VAR2.inh } VAR2' {VAR2.syn = VAR2'.syn}
VAR2 → ϵ {VAR2.syn = VAR2.inh}
PARAMLIST -> string {ident.type.inh = string} ident PARAMLIST2 {addType(ident.entry , type= ident.type.inh, ARRAY = null)}
PARAMLIST -> float {ident.type.inh = float} ident PARAMLIST2 {addType(ident.entry , type= ident.type.inh, ARRAY = null)}
PARAMLIST -> int {ident.type.inh = int} ident PARAMLIST2 {addType(ident.entry , type= ident.type.inh, ARRAY = null)}
PARAMLIST -> & {PARAMLIST .syn = PARAMLIST.inh}
PARAMLIST2 -> , {PARAMLIST.inh = PARAMLIST2.inh} PARAMLIST {PARAMLIST2.syn = PARAMLIST.syn}
PARAMLIST2 -> & {PARAMLIST2.syn = PARAMLIST2.inh}
ALLOCEXPRESSION -> new ALLOC2 {ALLOCEXPRESSION.type=ALLOC2.type ALLOCEXPRESSION.array= ALLOC2.array}
ALLOC2 -> string [NUMEXPRESSION] ALLOC3 {ALLOC2.type= string, ALLOC2.array = ALLOC3.syn) }
ALLOC2 -> float [NUMEXPRESSION] ALLOC3 {ALLOC2.type= float, ALLOC2.array = ALLOC3.syn) }
ALLOC2 -> int [NUMEXPRESSION] ALLOC3 {ALLOC2.type= int, ALLOC2.array = ALLOC3.syn) }
ALLOC3 -> [NUMEXPRESSION] ALLOC3` {ALLOC3.syn = ALLOC3'.syn}
ALLOC3 -> e {ALLOC3.syn = ALLOC.inh] }

2.3 VERIFICAÇÃO DE TIPOS

Aqui foi feito a operação de conferir todos os elementos até o primeiro ‘;’ seguido do valor que receberá atribuição. Caso todos os elementos sejam sinais aritméticos, comparações, ‘(’, ‘)’, ‘[’, ‘]’, ‘{’, ‘}’ o sinal de ‘=’ ou que possuam o mesmo tipo da variável que irá receber a atribuição será considerado um sucesso. Lembrando que os sinais ‘{’ e ‘}’ serão rejeitados no analisador sintático caso existam. Segue código abaixo.

var tipo = CurrentContext.PegaTipoDoSimbolo(token.s);

```
for (int j = i+1; j < lt.Count; j++)
{
        var tipo2 = CurrentContext.PegaTipoDoSimbolo(lt[j].s);
        if (    lt[j].a.Equals(Token.Attributes.ASSERT) |
                lt[j].a.Equals(Token.Attributes.ARITMETHIC) |
                lt[j].a.Equals(Token.Attributes.COMPARISON) |
                lt[j].a.Equals(Token.Attributes.BRKTPARE) |
                lt[j].t.Equals(tipo) |
                tipo.Equals(tipo2))
                        continue;
                if (lt[j].a.Equals(Token.Attributes.SEPARATOR))
                        break;
                SetErrorMessage(nt, lt[j].s, "Operação com atributos inválidos: ");
                return false;
        }
}
```

2.4 DECLARAÇÃO DE VARIÁVEIS POR ESCOPO

Foram criados contextos que possuem uma tabela de símbolos para os seguintes não terminais: FUNCDEF, STATEMENT, IF2, IFSTAT, FORSTAT e NUMEXPRESSION. Dentro da variável de contexto, é adicionado as propriedades se o escopo é ou não um loop (está dentro de um ou mais ‘for’), qual a variável esperada para ser o fim do contexto, e se o subcontexto pode ou não fechar o contexto do pai ao finalizar uma operação.

O atributo de contextos foi criado pensando em uma árvore que contém a tabela de símbolos, se o contexto é um loop, qual o terminal do contexto (para retornar para o contexto pai no algoritmo) e se ele pode fechar o contexto do pai dele (caso o não terminal dele seja o terminal do pai, acarretando em um ou mais fechamentos de contexto sucessivos).

Para checar se o atributo está no contexto, faz-se uma chamada recursiva para o contexto atual e o seu pai… Até a Raiz.

A adição de símbolos na tabela de símbolos primeiro checa se o símbolo já existe. Caso contrário adiciona no contexto atual.

2.5 COMANDOS DENTRO DE ESCOPOS

Para a verificação de ‘break’, foi usado a variável de Escopo. Dentro dela, se um break quiser ser adicionado, ele irá checar uma variável booleana dizendo que está dentro de um for. Caso não esteja, será recusado a entrada.

3. TAREFA GCI

Para concluir esta tarefa, foi construída uma SDT para geração de código em 3 endereços para a gramática CCC-2019-2.

3.1 CONSTRUIR UMA SDD L-ATRIBUÍDA PARA CCC-2019-2

Construção de uma SDD, contendo as produções já na forma normal de Chomsky, fatorada e sem recursão à esquerda.

Criamos as regras semânticas com o intuito da criação de um esquema de tradução para geração de códigos de *Three-address-code*. No geral, as produções propagam e/ou concatenam códigos que são gerados nas folhas para a raiz da árvore com as funções auxiliares como "*GENERATE*", "*CALL*", "*RETURN*", "*PRINT*" e "*READ*" e também com *labels* quando pertinente. Abaixo, segue o formato da SDD para a CCC-2019-2:

PROGRAM -> STATEMENT	PROGRAM.code = GENERATE(STATEMENT.code)	
PROGRAM -> FUNCLIST	PROGRAM.code = GENERATE(FUNCLIST.code)	
PROGRAM -> &	PROGRAM.code = GENERATE("")	
FUNCLIST -> FUNCDEF FUNCLIST2	FUNCLIST.code = FUNCDEF.code || FUNCLIST2.code	
FUNCLIST2 -> FUNCLIST	FUNCLIST2.code = FUNCLIST.code	
FUNCLIST2 -> &	FUNCTLIST2.code = ""	
FUNCDEF -> def ident (PARAMLIST) { STATELIST }	ident.entry = newLabel()	FUNCDEF.addr = ident.lexval FUNCDEF.code = label(ident.entry) || PARAMLIST .code || STATELIST.code ||

		GENERATE(RETURN())
PARAMLIST -> int ident PARAMLIST2	addType(ident.entry , type= ident.type.inh)	PARAMLIST.code = PARAMLIST2.code
PARAMLIST -> float ident PARAMLIST2	addType(ident.entry , type= ident.type.inh)	PARAMLIST.code = PARAMLIST2.code
PARAMLIST -> string ident PARAMLIST2	addType(ident.entry , type= ident.type.inh)	PARAMLIST.code = PARAMLIST2.code
PARAMLIST -> &	PARAMLIST .code = ""	
PARAMLIST2 -> , PARAMLIST	PARAMLIST2.code = PARAMLIST.code	
PARAMLIST2 -> &	PARAMLIST2 .code = ""	
STATEMENT -> VARDECL;	STATEMENT.addr = VARDECL.addr	STATEMENT.code = VARDECL.code VARDECL.next = STATEMENT.next
STATEMENT -> ATRIBSTAT;	STATEMENT.addr = ATRIBSTAT.addr	STATEMENT.code = ATRIBSTAT.code ATRIBSTAT.next = STATEMENT.next
STATEMENT -> PRINTSTAT;	STATEMENT.addr = PRINTSTAT.addr	STATEMENT.code = PRINTSTAT.code PRINTSTAT.next = STATEMENT.next
STATEMENT -> READSTAT;	STATEMENT.addr = READSTAT.addr	STATEMENT.code = READSTAT.code READSTAT.next = STATEMENT.next
STATEMENT -> RETURNSTAT;	STATEMENT.addr = RETURNSTAT.addr	STATEMENT.code = RETURNSTAT.code RETURNSTAT.next = STATEMENT.next
STATEMENT -> IFSTAT	STATEMENT.addr = IFSTAT.addr	STATEMENT.code = IFSTAT.code || STATEMENT.next STATEMENT.next = new Label()

		IFSTAT.next = STATEMENT.next
STATEMENT -> FORSTAT	STATEMENT.addr = FORSTAT.addr	STATEMENT.code = FORSTAT.code || STATEMENT.next STATEMENT.next = new Label() FORSTAT.next = STATEMENT.next
STATEMENT -> {STATELIST}	STATEMENT.addr = STATELIST.addr	STATEMENT.code = STATELIST.code
STATEMENT -> break ;	STATEMENT.code = GENERATE('goto' STATEMENT.next)	
STATEMENT -> ;	STATEMENT.addr = ""	STATEMENT.code = ""
ATRIBSTAT -> LVALUE = ATREXP	ATRIBSTAT.code = ATREXP.code || GENERATE(LVALUE.addr '=' ATREXP.addr)	LVALUE.addr = ATREXP.addr addType(ident.entry , type= ATREXP.type.inh, ARRAY = ATREXPRESSION.???)
ATREXP -> EXPRESSION	ATREXP.addr = EXPRESSION.addr	ATREXP.code = EXPRESSION.code
ATREXP -> ALLOCEXPRESSION	ATREXP.addr = ALLOCEXPRESSION.addr	ATREXP.code = ALLOCEXPRESSION.code
ATREXP -> FUNCCALL	ATREXP.addr = FUNCCALL.addr ATREXP.code = FUNCCALL.code	
FUNCCALL -> ident (PARAMLISTCALL)	FUNCALL.counter = 0 PARAMLIST.counter = FUNCALL.counter	PARAMLISTCALL .function_name = ident.lexval FUNCCAL.code= PARAMLISTCALL.code
PARAMLISTCALL -> ident PARAMLISTCALL2	PARAMLISTCALL.counter += 1 PARAMLISTCALL2.counter = PARAMLISTCALL.counter PARAMLISTCALL2.function_name =	PARAMLISTCALL.code = PARAMLISTCALL2.code PARAMLISTCALL.addr = PARAMLISTCALL2.addr

	PARAMLISTCALL .function_name	
PARAMLISTCALL -> &	PARAMLISTCALL .code = GENERATE(CALL(PARAMLISTCALL.function_name, PARAMLISTCALL.counter))	PARAMLISTCALL.addr = new Temp();
PARAMLISTCALL2 -> , PARAMLISTCALL	PARAMLISTCALL.counter = PARAMLISTCALL2.counter PARAMLISTCALL.function_name = PARAMLISTCALL2.function_name	PARAMLISTCALL2.code = PARAMLISTCALL.code PARAMLISTCALL2.addr = PARAMLISTCALL.addr
PARAMLISTCALL2 -> &	PARAMLISTCALL2.code = GENERATE(CALL(PARAMLISTCALL2.function_name, PARAMLISTCALL2.counter)))	PARAMLISTCALL2.addr = new Temp();
PRINTSTAT -> print EXPRESSION	PRINSTAT.code = GENERATE("print" EXPRESSION.addr)	
READSTAT -> read LVALUE	READSTAT.code = GENERATE(READ(LVALUE.addr))	
RETURNSTAT -> return	RETURNSTAT.code = GENERATE(RETURN())	
IFSTAT -> if(EXPRESSION) STATEMENT IF2	EXPRESSION.true = newlabel() EXPRESSION.false = newlabel()	IFSTAT.code = EXPRESSION.code || GENERATE("if" EXPRESSION.addr "goto" EXPRESSION.true) || GENERATE("goto" EXPRESSION.false) || label(EXPRESSION.true) || STATEMENT.code || GENERATE("goto" IFSTAT.next) || label(EXPRESSION.false) || IF2.code
IF2 -> else STATEMENT	IF2.code = STATEMENT.code	

IF2 -> &	IF2.code = ""	
FORSTAT -> for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT	EXPRESSION.true = newlabel() EXPRESSION.false = newlabel()	FORSTAT.code = ATRIBSTAT.code || EXPRESSION.code GENERATE("if" EXPRESSION.addr "goto" EXPRESSION.true) || GENERATE("goto" EXPRESSION.false) || label(EXPRESSION.true) || STATEMENT.code || GENERATE("goto" FORSTAT.next) || label(EXPRESSION.false)
STATELIST -> STATEMENT STATE2	STATELIST.code = STATEMENT.code || STATE2.code	
STATE2 -> STATELIST	STATE2.code = STATELIST.code	
STATE2 -> &	STATE2.code = ""	
EXPRESSION -> NUMEXPRESSION EXP2	EXP2.arg0 = NUMEXPRESSION.addr EXPRESSION.addr = EXP2.addr	EXPRESSION.code = NUMEXPRESSION.code || EXP2.code
EXP2 -> < NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '<' NUMEXPRESSION.addr)	
EXP2 -> > NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '>' NUMEXPRESSION.addr)	
EXP2 -> <= NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '<=' NUMEXPRESSION.addr)	
EXP2 -> >= NUMEXPRESSION	EXP2 .code = NUMEXPRESSION ||	

	GENERATE(EXP2.addr '=' EXP2.arg0 '>=' NUMEXPRESSION.addr)	
EXP2 -> == NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '==' NUMEXPRESSION.addr)	
EXP2 -> ! = NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '!=' NUMEXPRESSION.addr)	
EXP2 -> &	EXP2.code = ""	
NUMEXPRESSION -> TERM NUM2	NUM2.arg1 = TERM.addr	NUMEXPRESSION.addr = NUM2.addr NUMEXPRESSION.code = TERM.code || NUM2.code
NUM2 -> + TERM NUM2'	NUM2'.arg0 = NUM2.arg1 NUM2'.arg1 = NUM2.addr	NUM2.code = if (NUM2'.end) {gen(NUM2.addr '=' NUM2'.arg0 'plus' NUM2'.arg1)}else{gen(NUM2.addr '=' NUM2'.arg0 'plus' NUM2'.addr)}
NUM2 -> - TERM NUM2'	NUM2'.arg0 = NUM2.arg1 NUM2'.arg1 = NUM2.addr	NUM2.code = if (NUM2'.end) {gen(NUM2.addr '=' NUM2'.arg0 'minus' NUM2'.arg1)}else{gen(NUM2.addr '=' NUM2'.arg0 'minus' NUM2'.addr)}
NUM2 -> epson	NUM2.end= "true"	
TERM -> UNARYEXPR TERM2	TERM2.arg1 = UNARYEXPR.addr	TERM.addr = TERM2.addr TERM.code = UNARYEXPR.code || TERM2.code
TERM2 -> * UNARYEXPR TERM2'	TERM2'.arg0 = TERM2.arg1 TERM2'.arg1 =	TERM2.code = if (TERM2'.end)

	UNARYEXPR.addr	{gen(TERM2.addr '=' TERM2'.arg0 'multiply' TERM2'.arg1)}else{gen(TERM2.addr '=' TERM2'.arg0 'multiply' TERM2'.addr)}
TERM2 -> \ UNARYEXPR TERM2'	TERM2'.arg0 = TERM2.arg1 TERM2'.arg1 = UNARYEXPR.addr	TERM2.code = if (TERM2'.end) {gen(TERM2.addr '=' TERM2'.arg0 'divide' TERM2'.arg1)}else{gen(TERM2.addr '=' TERM2'.arg0 'divide' TERM2'.addr)}
TERM2 -> % UNARYEXPR TERM2'	TERM2'.arg0 = TERM2.arg1 TERM2'.arg1 = UNARYEXPR.addr	TERM2.code = if (TERM2'.end) {gen(TERM2.addr '=' TERM2'.arg0 'mod' TERM2'.arg1)}else{gen(TERM2.addr '=' TERM2'.arg0 'mod' TERM2'.addr)}
TERM2 -> e	TERM2.end = 'true'	
UNARYEXPR -> + FACTOR	UNARYEXPR.addr = new Temp()	UNARYEXPR.code = FACTOR.code || GENERATE(UNARYEXPR.addr '=' 'plus' FACTOR.addr)
UNARYEXPR -> - FACTOR	UNARYEXPR.addr = new Temp()	UNARYEXPR.code = FACTOR.code || GENERATE(UNARYEXPR.addr '=' 'minus' FACTOR.addr)
UNARYEXPR -> FACTOR	UNARYEXPR.addr = FACTOR.addr	UNARYEXPR.code = FACTOR.code
FACTOR -> int_constant	FACTOR.addr = int_constant.lexval	FACTOR.code = ""
FACTOR -> float_constant	FACTOR.addr = float_constant.lexval	FACTOR.code = ""
FACTOR -> string_constant	FACTOR.addr = string_constant.lexval	FACTOR.code = ""
FACTOR -> null	FACTOR.addr = null.lexval	FACTOR.code = ""
FACTOR -> LVALUE	FACTOR.addr = LVALUE .addr	FACTOR.code = LVALUE .code

FACTOR -> (NUMEXPRESSION)	FACTOR.addr = NUMEXPRESSION.addr	FACTOR.code = NUMEXPRESSION.code
ALLOCEXPRESSION -> new ALLOC2		
ALLOC2 -> int [NUMEXPRESSION] ALLOC3	ALLOC.code = NUMEXPRESSION.code || GENERATE(t '=' NUMEXPRESSION.addr '*' ALLOC2.array_id.type.width) GENERATE(ALLOC2.addr '= ALLOC3.addr '+' t)	ALLOC3'.array_id = ALLOC3.array_id t = new Temp() ALLOC3.addr = new Temp()
ALLOC2 -> float [NUMEXPRESSION] ALLOC3	ALLOC.code = NUMEXPRESSION.code || GENERATE(t '=' NUMEXPRESSION.addr '*' ALLOC2.array_id.type.width) GENERATE(ALLOC2.addr '= ALLOC3.addr '+' t)	
ALLOC2 -> string [NUMEXPRESSION] ALLOC3	ALLOC.code = NUMEXPRESSION.code || GENERATE(t '=' NUMEXPRESSION.addr '*' ALLOC2.array_id.type.width) GENERATE(ALLOC2.addr '= ALLOC3.addr '+' t)	
LVALUE -> ident ALLOC3		ALLOC3.array = false ALLOC3.array_id = ident.entry
ALLOC3 -> [NUMEXPRESSION] ALLOC3`		ALLOC3'.array = true ALLOC3'.array_id = ALLOC3.array_id t = new Temp() ALLOC3.addr = new Temp() GENERATE(t '=' NUMEXPRESSION.addr '*' ALLOC3.array_id.type.width) GENERATE(ALLOC3.addr '= ALLOC3'.addr '+' t)
ALLOC3 -> e	ALLOC3.code = ""	ALLOC3.addr= 0

VARDECL → string ident VAR2	addType(ident.entry , type= ident.type.inh, ARRAY = VAR2.)	ident.type= string VAR2.array = false VAR2.array_id = ident.entry
VARDECL → float ident VAR2	addType(ident.entry , type= ident.type.inh, ARRAY = VAR2.syn)	ident.type= float VAR2.array = false VAR2.array_id = ident.entry
VARDECL → int ident VAR2	addType(ident.entry , type= ident.type.inh, ARRAY = VAR2.syn)	ident.type= int VAR2.array = false VAR2.array_id = ident.entry
VAR2 → [int_constant] VAR2'		VAR2.array = true VAR2'.array_id = VAR2.array_id VAR2.addr = GENERATE(t '=' int_constant.lexval '*' VAR2.array_id.type.width) GENERATE(VAR2.addr '= VAR2'.addr '+' t) basta declarar no TS multidimensional.
VAR2 → e	VAR2.code = ""	VAR2.addr = 0

3.2 MOSTRAR QUE A SDD ANTERIOR É REALMENTE L-ATRIBUÍDA

Conforme mencionado anteriormente e aplicado novamente aqui, para mostrar que a SDD é do tipo L-atribuída, verificamos as produções que possuem atributos herdados, os quais devem ter origem do elemento pai ou de um irmão à esquerda. Abaixo um fragmento demonstrando o que ocorre em outras produções com herança.

EXPRESSION -> NUMEXPRESSION EXP2	EXP2.arg0 = NUMEXPRESSION.addr EXPRESSION.addr = EXP2.addr	EXPRESSION.code = NUMEXPRESSION.code || EXP2.code
EXP2 -> < NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '<' NUMEXPRESSION.addr)	
EXP2 -> &	EXP2.code = ""	

Neste fragmento, o resultado de NUMEXPRESSION é sintetizado em suas folhas e é herdado por EXP2 cujo qual utiliza em suas gerações de código.

3.3 CONSTRUIR UMA SDT PARA A SDD DE CCC-2019-2

// PROGRAM -> STATEMENT	PROGRAM.code = GENERATE(STATEMENT.code)	
// PROGRAM -> FUNCLIST	PROGRAM.code = GENERATE(FUNCLIST.code)	
// PROGRAM -> &	PROGRAM.code = GENERATE("")	
// FUNCLIST -> FUNCDEF FUNCLIST2	FUNCLIST.code = FUNCDEF.code || FUNCLIST2.code	
// FUNCLIST2 -> FUNCLIST	FUNCLIST2.code = FUNCLIST.code	
// FUNCLIST2 -> &	FUNCTLIST2.code = ""	
// FUNCDEF -> def ident (PARAMLIST) { STATELIST }	ident.entry = newLabel()	FUNCDEF.addr = ident.lexval FUNCDEF.code = label(ident.entry) || PARAMLIST .code || STATELIST.code || GENERATE(RETURN())

// PARAMLIST -> int ident PARAMLIST2	addType(ident.entry , type= ident.type.inh)	PARAMLIST.code = PARAMLIST2.code
// PARAMLIST -> float ident PARAMLIST2	addType(ident.entry , type= ident.type.inh)	PARAMLIST.code = PARAMLIST2.code
// PARAMLIST -> string ident PARAMLIST2	addType(ident.entry , type= ident.type.inh)	PARAMLIST.code = PARAMLIST2.code
// PARAMLIST -> &	PARAMLIST .code = ""	
// PARAMLIST2 -> , PARAMLIST	PARAMLIST2.code = PARAMLIST.code	
// PARAMLIST2 -> &	PARAMLIST2 .code = ""	
// STATEMENT -> VARDECL;	STATEMENT.addr = VARDECL.addr	STATEMENT.code = VARDECL.code VARDECL.next = STATEMENT.next
// STATEMENT -> ATRIBSTAT;	STATEMENT.addr = ATRIBSTAT.addr	STATEMENT.code = ATRIBSTAT.code ATRIBSTAT.next = STATEMENT.next
// STATEMENT -> PRINTSTAT;	STATEMENT.addr = PRINTSTAT.addr	STATEMENT.code = PRINTSTAT.code PRINTSTAT.next = STATEMENT.next
// STATEMENT -> READSTAT;	STATEMENT.addr = READSTAT.addr	STATEMENT.code = READSTAT.code READSTAT.next = STATEMENT.next
// STATEMENT -> RETURNSTAT;	STATEMENT.addr = RETURNSTAT.addr	STATEMENT.code = RETURNSTAT.code RETURNSTAT.next = STATEMENT.next
// STATEMENT -> IFSTAT	STATEMENT.addr = IFSTAT.addr	STATEMENT.code = IFSTAT.code || STATEMENT.next STATEMENT.next = new Label() IFSTAT.next =

		STATEMENT.next
// STATEMENT -> FORSTAT	STATEMENT.addr = FORSTAT.addr	STATEMENT.code = FORSTAT.code || STATEMENT.next STATEMENT.next = new Label() FORSTAT.next = STATEMENT.next
// STATEMENT -> {STATELIST}	STATEMENT.addr = STATELIST.addr	STATEMENT.code = STATELIST.code
// STATEMENT -> break ;	STATEMENT.code = GENERATE('goto' STATEMENT.next)	
// STATEMENT -> ;	STATEMENT.addr = ""	STATEMENT.code = ""
// ATRIBSTAT -> LVALUE = ATREXP	ATRIBSTAT.code = ATREXP.code || GENERATE(LVALUE.addr '=' ATREXP.addr)	LVALUE.addr = ATREXP.addr addType(ident.entry , type= ATREXP.type.inh, ARRAY = ATREXPRESSION.???)
// ATREXP -> EXPRESSION	ATREXP.addr = EXPRESSION.addr	ATREXP.code = EXPRESSION.code
// ATREXP -> ALLOCEXPRESSION	ATREXP.addr = ALLOCEXPRESSION.addr	ATREXP.code = ALLOCEXPRESSION.code
// ATREXP -> FUNCCALL	ATREXP.addr = FUNCCALL.addr ATREXP.code = FUNCCALL.code	
// FUNCCALL -> ident (PARAMLISTCALL)	FUNCALL.counter = 0 PARAMLIST.counter = FUNCALL.counter	PARAMLISTCALL .function_name = ident.lexval FUNCCAL.code= PARAMLISTCALL.code
// PARAMLISTCALL -> ident PARAMLISTCALL2	PARAMLISTCALL.counter += 1 PARAMLISTCALL2.counter = PARAMLISTCALL.counter PARAMLISTCALL2.function_name = PARAMLISTCALL	PARAMLISTCALL.code = PARAMLISTCALL2.code PARAMLISTCALL.addr = PARAMLISTCALL2.addr

	.function_name	
// PARAMLISTCALL -> &	PARAMLISTCALL .code = GENERATE(CALL(PARAMLISTCALL.function_name, PARAMLISTCALL.counter))	PARAMLISTCALL.addr = new Temp();
// PARAMLISTCALL2 -> , PARAMLISTCALL	PARAMLISTCALL.counter = PARAMLISTCALL2.counter PARAMLISTCALL.function_name = PARAMLISTCALL2.function_name	PARAMLISTCALL2.code = PARAMLISTCALL.code PARAMLISTCALL2.addr = PARAMLISTCALL.addr
// PARAMLISTCALL2 -> &	PARAMLISTCALL2.code = GENERATE(CALL(PARAMLISTCALL2.function_name, PARAMLISTCALL2.counter))	PARAMLISTCALL2.addr = new Temp();
// PRINTSTAT -> print EXPRESSION	PRINSTAT.code = GENERATE("print" EXPRESSION.addr)	
// READSTAT -> read LVALUE	READSTAT.code = GENERATE(READ(LVALUE.addr))	
// RETURNSTAT -> return	RETURNSTAT.code = GENERATE(RETURN())	
// IFSTAT -> if(EXPRESSION) STATEMENT IF2	EXPRESSION.true = newlabel() EXPRESSION.false = newlabel()	IFSTAT.code = EXPRESSION.code || GENERATE("if" EXPRESSION.addr "goto" EXPRESSION.true) || GENERATE("goto" EXPRESSION.false) || label(EXPRESSION.true) || STATEMENT.code || GENERATE("goto" IFSTAT.next) || label(EXPRESSION.false) || IF2.code
// IF2 -> else STATEMENT	IF2.code = STATEMENT.code	
// IF2 -> &	IF2.code = ""	

// FORSTAT -> for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT	EXPRESSION.true = newlabel() EXPRESSION.false = newlabel()	FORSTAT.code = ATRIBSTAT.code || EXPRESSION.code GENERATE("if" EXPRESSION.addr "goto" EXPRESSION.true) || GENERATE("goto" EXPRESSION.false) || label(EXPRESSION.true) || STATEMENT.code || GENERATE("goto" FORSTAT.next) || label(EXPRESSION.false)
// STATELIST -> STATEMENT STATE2	STATELIST.code = STATEMENT.code || STATE2.code	
// STATE2 -> STATELIST	STATE2.code = STATELIST.code	
// STATE2 -> &	STATE2.code = ""	
// EXPRESSION -> NUMEXPRESSION EXP2	EXP2.arg0 = NUMEXPRESSION.addr EXPRESSION.addr = EXP2.addr	EXPRESSION.code = NUMEXPRESSION.code || EXP2.code
// EXP2 -> < NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '<' NUMEXPRESSION.addr)	
// EXP2 -> > NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '>' NUMEXPRESSION.addr)	
// EXP2 -> <= NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '<=' NUMEXPRESSION.addr)	
// EXP2 -> >= NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '='	

	EXP2.arg0 '>=' NUMEXPRESSION.addr)	
// EXP2 -> == NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '==' NUMEXPRESSION.addr)	
// EXP2 -> ! = NUMEXPRESSION	EXP2 .code = NUMEXPRESSION || GENERATE(EXP2.addr '=' EXP2.arg0 '!=' NUMEXPRESSION.addr)	
// EXP2 -> &	EXP2.code = ""	
NUMEXPRESSION -> TERM NUM2	NUM2.arg1 = TERM.addr	NUMEXPRESSION.addr = NUM2.addr NUMEXPRESSION.code = TERM.code || NUM2.code
NUM2 -> + TERM NUM2'	NUM2'.arg0 = NUM2.arg1 NUM2'.arg1 = NUM2.addr	NUM2.code = if (NUM2'.end) {gen(NUM2.addr '=' NUM2'.arg0 'plus' NUM2'.arg1)}else{gen(NUM2.addr '=' NUM2'.arg0 'plus' NUM2'.addr)}
NUM2 -> - TERM NUM2'	NUM2'.arg0 = NUM2.arg1 NUM2'.arg1 = NUM2.addr	NUM2.code = if (NUM2'.end) {gen(NUM2.addr '=' NUM2'.arg0 'minus' NUM2'.arg1)}else{gen(NUM2.addr '=' NUM2'.arg0 'minus' NUM2'.addr)}
NUM2 -> epson	NUM2.end= "true"	
TERM -> UNARYEXPR TERM2	TERM2.arg1 = UNARYEXPR.addr	TERM.addr = TERM2.addr TERM.code = UNARYEXPR.code || TERM2.code
TERM2 -> * UNARYEXPR TERM2'	TERM2'.arg0 = TERM2.arg1 TERM2'.arg1 = UNARYEXPR.addr	TERM2.code = if (TERM2'.end) {gen(TERM2.addr '='

		TERM2'.arg0 'multiply' TERM2'.arg1)}else{gen(TERM2.addr '=' TERM2'.arg0 'multiply' TERM2'.addr)}
TERM2 -> \ UNARYEXPR TERM2'	TERM2'.arg0 = TERM2.arg1 TERM2'.arg1 = UNARYEXPR.addr	TERM2.code = if (TERM2'.end) {gen(TERM2.addr '=' TERM2'.arg0 'divide' TERM2'.arg1)}else{gen(TERM2.addr '=' TERM2'.arg0 'divide' TERM2'.addr)}
TERM2 -> % UNARYEXPR TERM2'	TERM2'.arg0 = TERM2.arg1 TERM2'.arg1 = UNARYEXPR.addr	TERM2.code = if (TERM2'.end) {gen(TERM2.addr '=' TERM2'.arg0 'mod' TERM2'.arg1)}else{gen(TERM2.addr '=' TERM2'.arg0 'mod' TERM2'.addr)}
TERM2 -> e	TERM2.end = 'true'	
UNARYEXPR -> + FACTOR	UNARYEXPR.addr = new Temp()	UNARYEXPR.code = FACTOR.code || GENERATE(UNARYEXPR.addr '=' 'plus' FACTOR.addr)
UNARYEXPR -> - FACTOR	UNARYEXPR.addr = new Temp()	UNARYEXPR.code = FACTOR.code || GENERATE(UNARYEXPR.addr '=' 'minus' FACTOR.addr)
UNARYEXPR -> FACTOR	UNARYEXPR.addr = FACTOR.addr	UNARYEXPR.code = FACTOR.code
FACTOR -> int_constant	FACTOR.addr = int_constant.lexval	FACTOR.code = ""
FACTOR -> float_constant	FACTOR.addr = float_constant.lexval	FACTOR.code = ""
FACTOR -> string_constant	FACTOR.addr = string_constant.lexval	FACTOR.code = ""
FACTOR -> null	FACTOR.addr = null.lexval	FACTOR.code = ""
FACTOR -> LVALUE	FACTOR.addr = LVALUE .addr	FACTOR.code = LVALUE .code
FACTOR -> (	FACTOR.addr =	FACTOR.code =

NUMEXPRESSION)	NUMEXPRESSION.addr	NUMEXPRESSION.code
// ALLOCEXPRESSION -> new ALLOC2		
// ALLOC2 -> int [NUMEXPRESSION] ALLOC3	ALLOC.code = NUMEXPRESSION.code || GENERATE(t '=' NUMEXPRESSION.addr '*' ALLOC2.array_id.type.width) GENERATE(ALLOC2.addr '= ALLOC3.addr '+' t)	ALLOC3'.array_id = ALLOC3.array_id t = new Temp() ALLOC3.addr = new Temp()
// ALLOC2 -> float [NUMEXPRESSION] ALLOC3	ALLOC.code = NUMEXPRESSION.code || GENERATE(t '=' NUMEXPRESSION.addr '*' ALLOC2.array_id.type.width) GENERATE(ALLOC2.addr '= ALLOC3.addr '+' t)	
// ALLOC2 -> string [NUMEXPRESSION] ALLOC3	ALLOC.code = NUMEXPRESSION.code || GENERATE(t '=' NUMEXPRESSION.addr '*' ALLOC2.array_id.type.width) GENERATE(ALLOC2.addr '= ALLOC3.addr '+' t)	
LVALUE -> ident ALLOC3	rvalue lvalue funcoes especiais no livro figura 2.44, 2.45	ALLOC3.array = false ALLOC3.array_id = ident.entry
ALLOC3 -> [NUMEXPRESSION] ALLOC3`		ALLOC3'.array = true ALLOC3'.array_id = ALLOC3.array_id t = new Temp() ALLOC3.addr = new Temp() GENERATE(t '=' NUMEXPRESSION.addr '*' ALLOC3.array_id.type.width) GENERATE(ALLOC3.addr '= ALLOC3'.addr '+' t)
ALLOC3 -> e	ALLOC3.code = ""	ALLOC3.addr= 0

VARDECL → string ident VAR2	addType(ident.entry , type= ident.type.inh, ARRAY = VAR2.)	ident.type= string VAR2.array = false VAR2.array_id = ident.entry
VARDECL → float ident VAR2	addType(ident.entry , type= ident.type.inh, ARRAY = VAR2.syn)	ident.type= float VAR2.array = false VAR2.array_id = ident.entry
VARDECL → int ident VAR2	addType(ident.entry , type= ident.type.inh, ARRAY = VAR2.syn)	ident.type= int VAR2.array = false VAR2.array_id = ident.entry
VAR2 → [int_constant] VAR2'		VAR2.array = true VAR2'.array_id = VAR2.array_id VAR2.addr = GENERATE(t '=' int_constant.lexval '*' VAR2.array_id.type.width) GENERATE(VAR2.addr '= VAR2'.addr '+' t) basta declarar no TS multidimensional.
VAR2 → e	VAR2.code = ""	VAR2.addr = 0

3.4 USAR A SDT DE CCC-2019-2 PARA GERAR CÓDIGO INTERMEDIÁRIO

Para essa fase, foi levado em consideração a geração de códigos em representação intermediária por três operadores. Todas operações numéricas são quebradas em instruções com três operandos, sendo gerados variáveis temporárias sob demanda. Essas instruções são anexadas aos atributos *code* dos não terminais e propagadas à raiz da gramática. Com a SDT também podemos identificar o momento em que os *labels* devem ser criados e onde devem

ser inseridos. Também é importante ressaltar a geração de instruções do tipo *goto*. Essas instruções são colocadas para o fluxo de controle explícito.

Também foram usadas funções *CALL* e *RETURN*, que implicitamente controlam o fluxo de instruções do programa, mantendo os endereços de retorno após chamadas de funções. O *CALL* especificamente armazena um endereço de retorno e como parâmetros recebe tanto o identificador da função como a quantidade de argumentos da função, que são colocados em pilha. Por sua vez, a função *RETURN* age como o par de completude à função *CALL*. Sua função é limpar os argumentos utilizados, que estão armazenados na pilha, e fazer o desvio para o endereço de retorno.

Por fim, a função *GENERATE*, é aplicada diretamente na geração dos códigos de três operandos, sintetizando expressões com variáveis temporárias. Além disso, é ela que gera nossas instruções de desvio, que tem papel importante na confecção de instruções *IF* e *FOR*.

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compiladores**: princípios, técnicas e ferramentas. Trad. de Daniel de Ariosto Pinto. Rio de Janeiro: LTC, 1995.

AHO, A. V.; LAM, M. S. SETHI, R.; ULLMAN, J. D. **Compilers**: principles, techniques and tools. 2. ed. Boston: Pearson; Addison-Wesley, 2007.

ULLMAN, J. D. **Styles of syntax-directed translations**. Disponível em: http://infolab.stanford.edu/~ullman/dragon/slides2.pdf . Acesso em: 5 dez. 2019.

GERMANO, A. **Lecture, GitHub.** Disponível em: https://github.com/germanoa/compiladores/tree/master/doc/lecture . Acesso em: 2 dez. 2019.