S -> {D}+{C}+

D -> VAR (integer [9] | char [10]) id [1][50][40] [D'] ';'[4] | CONST id [2] (= CONSTV [53][56][1]| '[num[33][54]]' = '"' string [48][55][2]'"') ';'

D'-> [= CONSTV [42][1]][4] {,id [1][51] [6] [[5]= CONSTV [42][1] | '[num[33][41][54]']'[3]][4]} | '[num[33][54][41]]'[3]{,id [1][51] [6] [[5]= CONSTV [42][1] | '[num[33][41][54]']'[3]][4]}

CONSTV -> 0x(hexa)(hexa) [44] | char [45] | Exp [43]

VALORCONST -> 0x(hexa)(hexa) [44] | char [45] | escalar | string [43]

C-> id [3][4] A ';'| FOR id [3][6][59] = E1[31][61] [28] [29] to E2[32][61] [step constanteID[3][36][?34]] do H | if Exp[35] [23] then J[25] | ';' | readln('id[3][6][9][[10]'[' Exp [33]']'][62] ')'';' | write('Exp[52] [20]{,Exp1[52}[20]')'';' | writeln('Exp[52] [20]{,Exp[52] [20]}')'';'[21]

A-> =[5] Exp [49][57][59] [1]| '['Exp']'[5][64][65] [7] = Exp [49][60] [8]

H-> C | '{' {C} '}'

J-> C [5][else[24] [6] ('{' {C} '}' || C)] | '{' {C} '}' [5][else[24] [6] ('{' {C} '}' || C)]

Exp-> $ExpS_1$ [7] [17]{[9] ('<' [8][66]| '>' [8][66] | '<=' [8][66] | '>=' [8][66] | '<>' [8][66] | '=' [10][66]) $ExpS_2$ [11][12][63][47] [18]}

ExpS-> [9] [+ [10]| -[10]] $ExpT_1$[14] [13] [15] {('+'[15][66] | '-' [16][66]| or [17][66]) $ExpT_2$ [18][19] [16]}

ExpT-> F_1 [20] [14]{('*' [21][66] | '/' [22][66] | '%' [23][66] | and [24][66]) F_2[25][26][19]}

F-> '(' Exp [27] ')' [12]| not F_1 [28] [13] | id [3][30] [9] ['['[38] Exp [39] '][58]'] [7]| VALORCONST [29] [11]

Ações Geração Código:

[5] [9] { cond = F } // Condição para comparações de entrada em trechos específicos

[6] [10] { cond =V } // Condição para comparações de entrada em trechos específicos

[1] { mov AX, DS:[exp.end]

mov DS:[id.end], AX}

[7] { F.end = NovoTemp

mov AX, DS:[id.end] // Endereco inicial do vetor

mov BX, DS:[Exp.end] // Endereco da expressao

se id.tipo == 'tipo_inteiro":

add BX,BX // Inteiros ocupam 2 bytes

add AX, BX // Posicao inicial do vetor + posicao desejada

mov DS:[F.end], AX

}

[8] {mov AX, Exp.getLexema()

mov DS:[endCalculado], AX}

[9] { F.end = id.end }

[10] { F.end = novoTemp

mov AX, DS:[endCalculado]

mov DS:[F.end], AX }

[11] { F.end = novoTemp

mov AX, num.lexema

mov DS:[F.end], AX }

[12] { F.end = Exp.end }

[13] { F.end = novoTemp

mov AX,DS:[1.end]

neg AX

add AX,1

mov DS:[F.end], AX}

[14] { ExpT.end = F.end }

[15] { se condicao para ser negativo então:

ExpS.end = novo temp

mov AX, DS:[ExpT1.end]

neg AX

mov DS:[ExpS.end], AX

senão:

ExpS.end = ExpT1.end}

[16] { ExpS.end = novo temp

mov AX, DS:[ExpT1.end]

mov BX, DS:[ExpT2.end]

RotFim := NovoRotulo

se operador = '+'

add AX, BX

jmp RotFim

senão se operador = '-'

sub AX, BX

jmp RotFim

senão se operador = 'or'

RotFim:

mov DS:[ExpS.end], AX}

[17] { Exp.end = ExpS1.end}

[18] { Exp.end = novo temp

```
mov AX, DS:[ExpS1.end]
mov BX, DS:[ExpS2.end]
cmp AX,BX
RotVerdadeiro := NovoRotulo
RotFim := NovoRotulo
se operador ‘<’ então:
jl RotVerdadeiro
mov Ax, 0
jmp RotFim
senão se  operador ‘>’ então:
jg RotVerdadeiro
mov Ax, 0
jmp RotFim
senão se operador ‘<=’
jle RotVerdadeiro
mov Ax, 0
jmp RotFim
senão se operador ‘>=’’
jge RotVerdadeiro
mov Ax, 0
jmp RotFim
senão se operador ‘<>’
jne RotVerdadeiro
mov Ax, 0
jmp RotFim
senão se operador ‘=’
je RotVerdadeiro
mov Ax, 0
jmp RotFim
senão se operador ‘<=’
mov Ax, 0
jmp RotFim
RotVerdadeiro:
```

mov AX, 1

RotFim:

mov DS:[Exp.end], AX}

[19] { ExpT.end = novo temp

mov AX, DS:[F1.end]

mov BX, DS:[F2.end]

RotFim := NovoRotulo

se operador = '*' então:

... no codigo

[20] {mov dx, Exp.end

mov ah, 09h

int 21h;"} // Printa o conteúdo do primeiro EXP

[21] {mov ah, 02h

int 21h;")} // Quebra de linha

[22] {mov dx, Exp1.end

mov ah, 09h

int 21h;")} // Printa o conteúdo dos EXPS seguintes, que foram separados por vírgula

[21] { mov AX, DS:[Exp.end]

mov BX, 1;

RotFalso: NewRot

cmp AX,BX

jne RotFalso

}

[24] {RotFalso:") }

[25] { RotFim:") }

[26] { logica do for: // Geracao completa implementada no codigo

Salvar o valor que vai iniciar o id do for em um temporario

Salvar o valor limite do for em outro temporario

Carregar os conteudos do id e do limite

Comparar: se igual pular fora do for, caso contario continua

Executa os comandos normalmente

Soma +1 no valor do temporario de id

Jump para a comparacao }

[27] { jmp RotFim }

[28] { RotInicio := NovoTemp, RotFim := NovoTemp }

[29] { mov AX, Exp.lexema

mov DS:[id.end], AX }

Ações semânticas:

[1] { se id.classe != vazio ERRO // Unicidade

senão id.classe = 'classe-var' } // Declaração da variável

[2] { se id.classe != vazio ERRO // Unicidade

senão id.classe = 'classe-const' } // Declaração de constante

[3] { se id.classe == vazio ERRO } // Identificador utilizado antes de declarar

[4] { A.classe = id.classe } // Atribuição recebe a classe do identificador

[5] { se A.classe != 'classe-var' ERRO } // Somente Variáveis podem receber um comando de atribuição

[6] { se id.classe != 'classe-var' ERRO } // Somente Variáveis podem receber um comando de atribuição

[7] { Exp.tipo = ExpS1.tipo } // O tipo de Exp é o tipo gerado por seu filho

[8] { se ExpS1.tipo != 'tipo-inteiro' OU Exps1.tamanho > 0 ERRO senão getOperador() } // String só aceita comparação de '='

[9] { cond = F } // Condição para comparações de entrada em trechos específicos

[10] { cond =V } // Condição para comparações de entrada em trechos específicos

[11] { se ExpS1.tipo != ExpS2. tipo OU Exps2.tamanho > 0 ERRO } // Verificar se podem ter tipos diferentes a serem comparados

[12] { se ExpS1.tipo == 'tipo-string' && cond = F ERRO } // Comparando duas strings sem ser por igualdade

[13] { se ExpT1.tipo != 'tipo-inteiro' && cond = V ERRO } // Colocando sinais (+ ou -) em não inteiros

[14] { ExpS.tipo = ExpT1.tipo } // ExpS é do tipo gerado por seu filho

[15] { se ExpT1.tipo != 'tipo-inteiro' ERRo

senão operador = '+' } // Operador utilizado

[16] { se ExpT1.tipo != 'tipo-inteiro' ERRO

senão operador = '-' } // Operador utilizado

[17] { se ExpT1.tipo != 'tipo-logico' ERRO

senão operador = 'or' } // Operador utilizado

[18] { se ExpT1.tipo != ExpT2.tipo ERRO } // Verificar se existem tipos que podem ser comparaveis

[19] { se ExpT2.tipo != 'tipo-lógico' && operador = = 'or' ||

Expt2.tipo != 'tipo-inteiro' && operador == '+' || operador == '-' ERRO } // + e - somente com tipo int

[20] { ExpT.tipo != F1.tipo } // ExpT possui o tipo do seu filho

[21] { se F1.tipo != 'tipo-inteiro' ERRO

senão operador = ‘*’ } // Operador utilizado

[22] { se F1.tipo != ‘tipo-inteiro’ ERRO

senão operador = ‘/’ } // Operador utilizado

[23] { se F1.tipo != ‘tipo-inteiro’ ERRO

senão operador = ‘%’ } // Operador utilizado

[24] { se F1.tipo != ‘tipo-logico’ ERRO

senão operador = ‘AND’ } // Operador utilizado

[25] { se F1.tipo != F2.tipo ERRO } // Verificar se existem tipos que podem ser comparaveis

[26] { se F2.tipo != ‘tipo-lógico’ && operador = = ‘AND’ ||

F2.tipo != ‘tipo-inteiro’ && operador == ‘*’ || operador == ‘/’ || operador == ‘%’ ERRO } //

[27] { F.tipo = E.tipo} // o tipo do E vai ser o mesmo do F pq está concatenando uma lista de exps

[28] { F1.tipo != logico então ERRO }

[29] { F.tipo = VALORCONST.tipo }

[30] { F.tipo = id.tipo }

[31] { se E1.tipo != inteiro || E1.tipo != id.tipo então ERRO }

[32] { se E2.tipo != inteiro || E1.tipo != id.tipo então ERRO }

[33] { se E.tipo != ‘tipo-inteiro’ então ERRO }

[34] { se E.valor < 1 então ERRO }

[35] { se E3.tipo != logico então ERRO }

[36] { se !(E.tipo != “tipo_inteiro” && E.classe != “‘classe-const’ ”) então ERRO }

[37] { se E.tamanho > 0 então ERRO }

[38] { se id.getTamanho() <= 0 ERRO }

[39] { se E.tipo != ‘tipo-inteiro’ ERRO

senão se E.valor > id.tamanho ERRO }

[40] { D’.tipo = id.tipo }

[41] { se D’.tipo == ‘tipo-inteiro’

se E.valor > TamanhoMaximoVetorInteiro ERRO

senão se D’.tipo == ‘tipo-char’

se E.valor > TamanhoMaximoVetorChar Erro

senão ERRO } // senão ERRO → Var != inteiro ou char

[42] { se D’.tipo != CONSTV.tipo ERRO }

[43] { CONSTV.tipo = E.tipo }

[44] { CONSTV.tipo = ‘tipo-caracter’ }

[45] { CONSTV.tipo = ‘tipo-char’ }

[47] { Exp.tipo = ‘tipo-logico’ }

[48] { se E.valor < string.length + 1 ERRo } // Verificar se a string cabe no vetor de caracteres

[49] { A.tipo = E.tipo }

[50] { se cond então

id.tipo = “tipo_char”

senão id.tipo = “tipo_inteiro”

}

[51] { id.tipo = D’.tipo }

[52] { se exp.tipo != “tipo_inteiro” && exp.tipo != “tipo_char” && exp.tipo != “tipo_string” }

[53] { id.tipo = constv.tipo }

[54] { se E.tipo = “tipo_inteiro”

então id.tamanho = E.valor

senão ERRO}

[55] { id.tipo = “tipo_string”}

[56] { se id.tamanho <= 0 && constv.tipo = string ERRO}

[57] { se E.tipo = “tipo_string” && id.tipo = “tipo_caracter”

se E.valor > id.tamanho ERRO

senao se id.tipo != E.tipo ERRO}

[58] {id.tamanho = 0}

[59] { se id.getTamanho > 0 então erro }

[60] { (se id.getTipo = ‘tipo_caracter’ && E.getTipo = ‘tipo_string’) OU

(se id.getTipo = ‘tipo_inteiro’ && E.getTipo = ‘tipo_string’) então ERRO}

[61] { se E.getTamanho > 0 }

[62] { se cond = F && id.getTamanho > 0 ERRO }

[63] { se cond V

se exps1.getTipo = inteiro & exps1.tamanho > 0 || exps2.getTipo = inteiro & exps2.tamanho > 0 entao ERRO }

[64] { se E.getTipo = ‘tipo_inteiro’

se E.getTamanho > id.getTamanho entao ERRO }

[65] { se F.id.getTamanho == 0 ERRO }

[66] { getOperador() }