Trabalho de Compiladores

Verificação de unicidade, classes e tipos

Geração de código

Alunos:

Ana Letícia Camargos
Augusto Noronha
Cora Silberschneider

1. Gramática

S → {**D**}* {**C**}* eof

D → var { (char | integer) id [= valor | '[' valor ']'] {, id [= valor | '[' valor ']'] } ;
}$^{+}$ |
const id = valor ;

C → id ('[' **Exp** ']' = **Exp**; | = **Exp**;) |
For id = **Exp** to **Exp** [step valor] do **B** |
if **Exp** then **C** [else ('{' {**C**} '}' | **C**)] | '{' {**C**} '}' [else ('{' {**C**} '}' | **C**)] |
; |
readln'(' id ')'; |
write'(' **Exp** {, **Exp**} ')'; |
writeln'(' **Exp** {, **Exp**} ')';

B → **C** | '{' {**C**} '}'

Exp → **ExpS** [(= | <> | < | > | <= | >=) **ExpS**]
ExpS → [+ | -] **T** { (+ | - | or) **T** }
T → **F** { (* | and | / | %) **F** }
F → not **F** | valor | id ['[' **Exp** ']'] | '(' **Exp** ')'

2. Esquema de Tradução

S → {**D**}* {**C**}* eof

D → var { [32] (char [33] | integer [34]) id [23][35]
[(= [45] [- [44]] valor [38] [36] | '[' valor [20] ']')
{, id [23][35] [(= [45] [- [44]] valor [38] [36]
| '[' valor [20] ']')] } ; } + |
const id [21] = [45] [- [44]] valor [38] [22] [49] ;

C → id [24] ('[' **Exp1** [40] ']' = **Exp2** [41] ; | = **Exp** ; [42][43]) |
For id [24] [29] = **Exp** [27] [26] to **Exp** [27] [step valor [25]] do **B** |
if **Exp** [28] then (**C** | '{' **C** '}') [else ('{' {**C**} '}' | **C**)] |
; |
readln'(' id [24] [31] ')'; |
write'(' **Exp** [30] {, **Exp** [30] } ')'; |
writeln'(' **Exp** [30] {, **Exp**} [30] ')';

B → **C** | '{' {**C**} '}'

Exp → **ExpS1** [17][47] [(= [46] | <> [19] | < [19] | > [19] | <= [19] | >= [19]) **ExpS2** [18]]

ExpS → [10] [+ [11] | - [12]] **T1** [13] {[48] (+ [11] | - [12] | or [16]) **T2** [14] }

T → **F1** [6] { (* [7] | and [8] | / [7] | % [7]) **F2** [9] }
F → not **F1** [4] | valor [3] | id [24][1] ['[' **Exp** ']' [5]] | '(' **Exp** [2] ')'

Verificação de Tipos

D → [21] { se id.classe != vazio então ERRO
senão id.classe = const }
[22] { id.tipo = valor.tipo }
[23] { se id.classe != vazio então ERRO
senão id.classe = var }
[32] { condChar = false

condInteger = false}
[33] { condChar = true }
[34] { condInteger = true }
[35] { se condChar = true então
id.tipo = caractere
se condInteger = true então
id.tipo = inteiro}
[44] {condNeg = True}
[45] {condNeg = False}
[38] {se condNeg então
se valor.tipo != inteiro então ERRO}

[36] { se id.tipo != valor.tipo então ERRO }
[20] { se valor.tipo != inteiro então ERRO
se contInteger E valor.lex > 1024 então ERRO
se condChar E valor.lex > 4096 então ERRO
id.tipo = arranjo(valor.lex, id.tipo)}
[49] { se valor.tipo != inteiro E valor.tipo != caractere então ERRO }

C → [24] { se id.classe = vazio então ERRO }
[25] { se valor.tipo != inteiro então ERRO }
[26] { se id.tipo != Exp.tipo então ERRO }
[27] { se Exp.tipo != inteiro então ERRO }
[28] { se Exp.tipo != lógico então ERRO }
[29] { se id.tipo != inteiro então ERRO }
[30] { se Exp.tipo != inteiro E Exp.tipo != caracter E Exp.tipo != arranjo(i, caractere)
então ERRO }
[31] { se id.tipo != inteiro E id.tipo != caracter E id.tipo != arranjo(i, caractere)(string) então ERRO }
[40] { se Exp1.tipo != inteiro então ERRO
senão se id.tipo != arranjo(i, j) então ERRO}
[41] { se id.tipo != arranjo(i, Exp2.tipo) então ERRO }
[42] { se id.tipo != Exp.tipo então ERRO
[43] { se id.tipo = arranjo(i, j) então
se Exp.tipo = string E id.tamanho + 1 < exp.tamanho então ERRO

se Exp.tipo = arranjo(i, j) E id.tamanho< exp.tamanho então ERRO
}

Exp → **[17]** { Exp.tipo = ExpS.tipo }
[19] {condIgualdade = False}
[47] {condIgualdade = False}
[46] {condIgualdade = True}
[18] {se ExpS1.tipo != ExpS2.tipo então ERRO
se condIgualdade = FALSE então
se ExpS1.tipo = arranjo(i, t) então ERRO
Exp.tipo = Logico}

ExpS → **[10]** { condMais = false
condMenos = false}
[11] { condMais = true }
[12] { condMenos = true }
[13] { se condMais = true OU condMenos = true então
se T1.tipo != inteiro E T1.tipo != caractere então ERRO
ExpS.tipo = T1.tipo }
[48] { condMais = false
condMenos = false
condLog = false}
[16] {condLog = True}
[14] { se T1.tipo != T2.tipo então ERRO
se condLog então
se T1.tipo != lógico então ERRO
se condMais ou condMenos então
se T1.tipo != caractere E T1.tipo != inteiro ERRO

T → **[6]** { T.tipo = F1.tipo }
[7] { se F1.tipo != inteiro então ERRO }
[8] { se F1.tipo != lógico então ERRO }
[9] { se F1.tipo != F2.tipo então ERRO }

F → **[1]** { F.tipo = id.tipo }
[2] { F.tipo = Exp.tipo }
[3] { F.tipo = valor.tipo
se valor.tipo = String então F.tamanho = valor.tamanho }

[4] { se F1.tipo != lógico então ERRO
senão F.tipo = F1.tipo }
[5] { se Exp.tipo = inteiro E id.tipo = arranjo(i, t) então
F.tipo = t senão ERRO }
[24] { se id.classe = vazio então ERRO }

Geração de Código

S → {**D**}* {**C**}* eof

D → var { (char | integer) id [**[38]** = [-] valor **[41]** '[' valor ']' **[42]**] **[25]** {, id [**[38]** = [-] valor **[37]** **[41]** | '[' valor ']' **[42]**] } ; }$^{+}$ |
const id = [-] valor **[37]** **[41]** ;

C → id ('[' **Exp1** ']' = **Exp2**; **[28]** | = **Exp2**; **[27]**)
For id = **Exp3** **[30]** to **Exp4** **[31]** [step valor] do **B** **[32]** **[33]**|
if **[34]** **Exp5** **[35]** then (C | '{' C '}') **[36]**[45] [else ('{' {C } '}' | C)]**[46]**|
; |
readln(' id ');**[44]** |
write(' **Exp** **[43]**{, **Exp** **[43]**} '); |
writeln(' **Exp** **[43]**[47] {, **Exp** **[43]** **[47]**} ');

B → **C** | '{' {**C**} '}'

Exp → **ExpS1** **[17]** [(= **[18]** | <> **[19]** | < **[20]**| > **[21]**| <= **[22]** | >= **[23]**) **ExpS2** **[24]**]

ExpS → **[40]** [+ | - **[39]**] **T1** **[12]** { (+ **[13]** | - **[14]** | or **[15]**) **T2** **[16]** }

T → **F1** **[6]**{ (* **[8]** | and **[9]** | / **[10]** | % **[11]**) **F2** **[7]**}
F → not **F1** **[5]** | valor **[2]** | id **[1]** ['[' **Exp** ']' **[3]**] | '(' **Exp** ')' **[4]**

[1] { F.end := id.end }
[2] { sword valor.lex
F.end := NovoTemp
mov Ax, valor.lex
mov DS:[F.end], Ax}
[3] { mov Ax, DS:[Exp.end]
add Ax, Ax ; se char nao add

```
        add Ax, id.end
        mov Ax, DS:[Ax]
        mov DS:[F.end], Ax }
[4] { F.end := Exp.end }
[5]  { mov Ax, DS:[F1.end]
      not Ax
      F.end = NovoTemp
      mov DS:[F.end], Ax}
[6] { T.end := F1.end
       condMul = False
       condAnd = False
       condDiv = False
       condMod = False}
[8] {condMul = True}
[9] {condAnd = True}
[10] {condDiv = True}
[11] {condMod = True}
[7]   { mov Ax, DS:[T.end]
        mov Bx, DS:[F2.end]
        T.end := NovoTemp
        se condMul então:
                mul Ax, Bx
                mov DS:[T.end], Ax
        senão se condAnd então:
                and Ax, Bx
                mov DS:[T.end], Ax
        senão se condDiv então:
                cwd
                idiv Bx
                mov DS:[T.end], Bx
        senão se condMod então:
                mov Dx, 0  #guarda o resto da divisão
                cwd
                idiv Bx
                mov DS:[T.end], Dx}
[12] {ExpS.end := T1.end
       se condMenos então
                mov Ax, DS:[ExpS.end]
```

mul Ax, -1
mov DS:[ExpS.end], Ax
condMenos = false }

[40] { condMenos = false}

[39] { condMenos = true}

[13] {condMais := True}

[14] {condMenos := True}

[15] {condOr := True }

[16] { mov Ax, DS:[ExpS.end]
mov Bx, DS:[T2.end]
ExpS.end := Novo Temp
se condMais então:
add Ax, Bx
mov DS:[ExpS.end], Ax
senão se condMenos então:
sub Ax, Bx
mov DS:[ExpS.end], Ax
senão se condOr então:
or Ax, Bx
mov DS:[ExpS.end], Ax}

[17] { Exp.end := ExpS1.end
condIgual = false
condDif = false
condMenor = false
condMaior = false
condMenorIgual = false
condMaiorIgual = false }

[18] {condIgual := True}

[19] {condDif := True}

[20] {condMenor := True}

[21] {condMaior := True}

[22] {condMenorIgual := True}

[23] {condMaiorIgual := True}

[24] {mov Ax, DS:[ExpS.end]
cwd ; *converter ambos para inteiro, atribuindo 0 ao registrador alto*
mov Bx, DS:[ExpS2.end]
cwd ; *converter ambos para inteiro, atribuindo 0 ao registrador alto*
RotVerdadeiro := NovoRot

```
        cmp Ax, Bx
        se condIgual então:
                je RotVerdadeiro
        se condDif então:
                jne RotVerdadeiro
        se condMenor então:
                jl RotVerdadeiro
        se condMaior então:
                jg RotVerdadeiro
        se condMenorIgual então:
                jle RotVerdadeiro
        se condMaiorIgual então:
                jge RotVerdadeiro

        mov Ax, 0 ; copia o valor 0 para retornar false
        RotFim := NovoRot
        jmp RotFim
RotVerdadeiro:
        mov Ax, 1  ; copia o valor 1 para retornar true
RotFim:
        Exp.end := NovoTemp
        Exp.tipo := lógico
        mov DS:[Exp.end], Ax } // ds:[Exp.end]?

[28] { mov Ax, DS:[Exp1.end]
          mov Bx, DS:[Exp2.end]
          add Ax, Ax  ; se inteiro faz essa linha, se char nao faz
          add Ax, id.end
          mov DS:[Ax], Bx } ; movendo para dentro de Ax o que tem em Exp2

[27]    { mov Ax, DS:[Exp2.end]
        mov DS:[id.end], Ax} }

[30] { mov Ax, DS:[Exp3.end]
        mov DS:[id.end], Ax
        RotInico := NovoRot
        RotFim := NovoRot}
```

[31] {RotInicio:
mov Ax, DS:[id.end]
mov Bx, DS:[Exp4.end] ; *carrega conteúdo de Exp4*
cmp Ax, Bx ; *compara o conteúdo de id, se id > Exp desvia para RotFim*
jg RotFim}
[32] { mov Ax, DS:[id.end]
add Ax, valor.lex ; *incrementa id*
mov DS:[id.end], Ax
jmp RotInicio}
[33] { RotFim: }
[34] { RotFalso := NovoRot
RotFim := NovoRot}
[35] { mov Ax, DS:[Exp5.end]
mov Bx, 0 ; *equivale a um booleano igual a 0*
cmp Ax, Bx ; *checa se Ax é falso*
je RotFalso}
[36] { RotFalso: }
[37] { alocar id.end novoEnd }
[38] {condEntrou := True}
[25] {se condEntrou == False então
se condChar então
byte ?
se condInteger então
sword?}

[41] {se condNeg então:
valor.lex = - valor.lex
sword valor.lex}
[42] {byte valor.lex DUP(?)}

[45] {jmp RotFim}
[46] {RotFim:}
[44] { C.end := novoTemp
Rot1 := novoRot
…}

```
[43] { se Exp.tipo == string então
            mov Dx, Exp.end
            mov ah, 09h
            int 21h
       senão se Exp.tipo == caractere então
            mov Al, DS:[Exp.end]
            Exp.end := novoEnd
            mov DS:[Exp.end], Al
            mov Bl, 24h
            mov DS:[Exp.end +1], Bl
            mov Dx, Exp.end
            mov ah, 09h
            int 21h
       senão se Exp.tipo == string então
             mov Dx, Exp.end
            mov ah, 09h
            int 21h
       senão se Exp.tipo == inteiro então
            mov Di, 0 ; contador
            mov Ax, DS:[Exp.end]
            Exp.end := novoTemp
            mov Di, Exp.end
            cmp Ax, 0 ; verifica sinal
            Rot1 := novoRot
            jge Rot1 ;
            mov Bl, 2Dh ;senão escreve sinal -
            mov DS:[Di], Bl
            mov Di, 1 ; incrementa indice
            neg ax ; toma modulo do numero
        Rot1:
            mov bx, 10 ; divisor
            Rot2 := novoRot
       Rot2:
            add Cx, 1 ; incrementa contador
            mov Dx, 0
            idiv bx
            push dx
            cmp ax, 0
```

```
            jne Rot2
            Rot3 := novoRot
        Rot3:
            pop Dx
            add Dx, 30h
            mov DS:[di], Dl
            add Di, 1
            add, cx, -1
            cmp Cx, 0
            jne Rot3
            mov Dl, 024h
            mov DS:[di], dl

            mov dx, C.end
            mov Ah, 09h
            int 21h }
[47]    {mov ah, 02h
        mov dl, 0Dh
        int 21h
        mov DL, 0Ah
        int 21h}
```