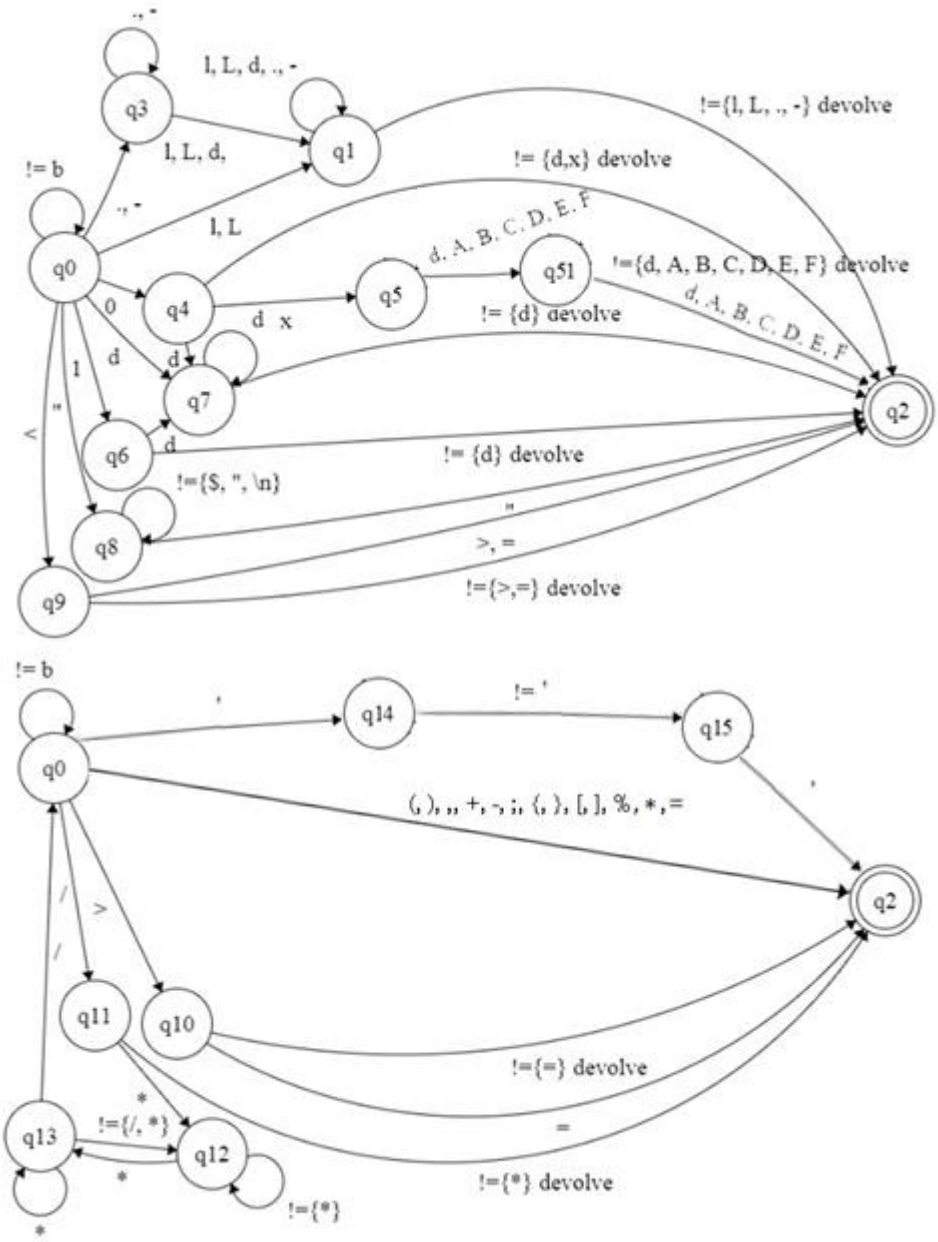
Arthur Floresta, Marcos Felipe Vendramini, Pedro Sodré

Relatório Final

Token	Lexema
Id	([(l/L)(l/L/d/./_)*]/[(./_)(l/L/d/./_)+])
Const	(C/c)(O/o)(N/n)(S/s)(T/t)
Var	(V/v)(A/a)(R/r)
Integer	(I/i)(N/n)(T/t)(E/e)(G/g)(E/e)(R/r)
Char	(C/c)(H/h)(A/a)(R/r)
For	(F/f)(O/o)(R/r)
If	(I/i)(F/f)
Else	(E/e)(L/l)(S/s)(E/e)
And	(A/a)(N/n)(D/d)
Or	(O/o)(R/r)
Not	(N/n)(O/o)(T/t)
To	(T/t)(O/o)
=	=
(	)
)	(
>	>
<	<
+	+
-	-
/	/
{	{
}	}
%	%
[	[
]	]
Then	(T/t)(H/h)(E/e)(N/n)
Readln	(R/r)(E/e)(A/a)(D/d)(L/l)(N/n)
Step	(S/s)(T/t)(E/e)(P/p)
Write	(W/w)(R/r)(I/i)(T/t)(E/e)
Writeln	(W/w)(R/r)(I/i)(T/t)(E/e)(L/l)(N/n)
Do	(D/d)(O/o)



Gramática

S -> [D] [CO] {AT| R| TES| NU| L| E}+

D -> Var A

A -> {integer Z; | char W; }+

Z -> I K

W -> I C

K -> [,Z] | [= [-] constante[,Z]]

C -> [,W] | [= constante[,W]]

CO -> const X;

X -> Y {,Y}

Y -> I = [-] constante

AT -> I = EXP;

R -> for I = EXP to EXP [step constante] do P

P -> S | “{“ S “}”

TES -> if EXP then P [else P]

NU -> ;

L -> readln “(“ I “)”;

E -> (write “(“ EXP {,EXP} “)”; | writeln “(“ EXP {,EXP} “)”;)

EXP -> EXPS [(= | <>| < | >| >= | <=) EXPS]

EXPS -> [+| -] T {(+| -| or)T}

T -> F {(*|and| / | %) F}

F -> N {(not) N}

N -> (I| constante| “(“ EXP “)”)

I -> id [“[“constante”]”]

Verificação de tipo

Função I

id.tipo=I.tipo;

If (cons.tipo != inteiro) erro;

Else{

    If(I.tipo == inteiro) id.tipo==arranjo(num.lex,inteiro);

    Else if (I.tipo == caractere) id.tipo==arranjo(num.lex,caractere);

    }

Função N

N.tipo= I.tipo;

N.tipo= const.tipo;

N.tipo= EXP.tipo;

Função F

F.tipo = N.tipo;

if(N1.tipo != N2.tipo) ERRO;

If (F.tipo != logico) ERRO;

Função T

T.tipo = F.tipo;

If (F1.tipo !=F2.tipo) ERRO;

Else if (F1.tipo == caractere) ERRO;

Else T.tipo = F.tipo;

Função EXPS

EXPS.tipo = T.tipo;

If (T1.tipo != T2.tipo) erro;

Else if ((aux==”-“||aux ==”or”)&&T1.tipo==caractere) erro;

Else EXPS.tipo = T1.tipo;

Função EXP

EXP.tipo = EXPS.tipo;

If (EXPS1.tipo != EXPS2.tipo) erro;

Else if (EXPS1.tipo == char && (aux !=”=”|| aux !=”<>”) erro;

Else EXP.tipo = EXPS.tipo;

Função TES

If (Exp.tipo !=logico) erro;

Função R

If(I.tipo != EXP1.tipo || I.tipo!=EXP2.tipo) erro;

Função AT

If (I.tipo!= EXP.tipo) erro;

Função Y

I.tipo = const.tipo;

Função X

X.tipo = Y.tipo;

Função CO

If (X.tipo != const.tipo) erro;

Z.class = var;

Z.tipo = inteiro;

I.tipo = Z.tipo;

K.tipo = I.tipo;

Função W

W.class = var;

W.tipo = caractere;

I.tipo = Z.tipo;

C.tipo = I.tipo;

Função K

K.tipo == Z.tipo

If(K.tipo != const.tipo) erro;

Função C

If (C.tipo != W.tipo)erro;

If (C.tipo != const.tipo) erro;

Geração de Código

Função S -> 1 [D] [CO] 2 {AT| R| TES| NU| L| E} + 3

1

sseg SEGMENT STACK

byte 4000h DUP(?)

sseg ENDS

dseg SEGMENT PUBLIC

byte 4000h DUP(?)

2

dseg ENDS

cseg SEGMENT PUBLIC

ASSUME CS:cseg, DS:dseg

strt:

3

cseg ENDS

END strt

Função Z

sword ?

Função W

byte ?

Função I

If (constante exists)

    If (I.tipo==inteiro)

        Sword “valor constante”-1 DUP(?)

    If (I.tipo==char)

        Byte ”valor constante”-1 DUP(?)

Função Y

If (inteiro) Sword ?

If (caractere) Byte ?

Mov DS:[id.end], valor constante

Função K

Mov DS:[id.end], valor constante

Função C

Mov DS:[id.end], valor constante

Função AT

Mov DS:[id.end], EXP.value

Função R -> for I = EXP to EXP 1 [step constante 2] do P 3

1

Mov DS:[id.end], Exp1.value

Mov Ax, Exp2.value

IniFor:

Mov BX, DS:[id.end]

Cmp ax,bx

Je FimFor

2

Add bx, step

Mov DS:[id.end], bx

3

FimFor:

Função TES -> if EXP 1 then 2 P [else P 3] 4

1

Mov bx, EXPval

Cmp bx,1

2

Jne RotFalso

Jmp rotFim

3

RotFalso:

4

rotFim:

Função L

mov dx, buffer.end

mov al, 0FFh

mov ds:[buffer.end], al

```
mov ah, 0Ah
int 21h
if (id.tipo = arranjo(num,char)){
mov di, buffer.end+2
mov ax, 0
mov cx, 10
mov dx, 1
mov bh, 0
mov bl, ds:[di]
cmp bx, 2Dh
jne R0
mov dx, -1
add di, 1
mov bl, ds:[di]
R0:
push dx
mov dx, 0
R1:
cmp bx, 0dh
je R2
imul cx
add bx, -48
add ax, bx
add di, 1
mov bh, 0
mov bl, ds:[di]
jmp R1
R2:
pop cx
imul cx
mov DS:[id_end], AX
}else{
mov di, buffer.end+2
```

```
mov bl, ds:[di]
mov DS:[id_end], bl
}
```

Função E -> (write “(“ EXP {,EXP} “)”; 1 | writeln “(“ EXP {,EXP} “)”; 2)

```
1
Mov di, string.end
mov cx, 0
cmp ax,0
jge R0
mov bl, 2Dh
mov ds:[di], bl
add di, 1
neg ax
R0:
mov bx, 10
R1:
add cx, 1
mov dx, 0
idiv bx
push dx
cmp ax, 0
jne R1
R2:
pop dx
add dx, 30h
mov ds:[di], dl
add di, 1
add cx, -1
cmp cx, 0
jne R2
mov dl, 024h
mov ds:[di], dl
```

```
mov dx, string.end
mov ah, 09h
int 21h
2
mov di, string.end
mov cx, 0
cmp ax,0
jge R0
mov bl, 2Dh
mov ds:[di], bl
add di, 1
neg ax
R0:
mov bx, 10
R1:
add cx, 1
mov dx, 0
idiv bx
push dx
cmp ax, 0
jne R1
R2:
pop dx
add dx, 30h
mov ds:[di], dl
add di, 1
add cx, -1
cmp cx, 0
jne R2
mov dl, 024h
mov ds:[di], dl
mov dx, string.end
mov ah, 09h
```

```
int 21h
mov ah, 02h
mov dl, 0Dh
int 21h
mov DL, 0Ah
int 21h
```

Função EXP -> EXPS [(= 1 | <> 2| < 3 | > 4| >= 5| <= 6) EXPS 7]

```
1
Mov ax, ds:[exps1.end]
Mov bx, ds:[exps2.end]
Cmp ax,bx
Je truexp
Mov ax,0;
Jmp fimexp
Truexp:
Mov ax,1;
Fimexp:
Mov ds:[exp.end], ax
2
Mov ax, ds:[exps1.end]
Mov bx, ds:[exps2.end]
Cmp ax,bx
Jne truexp
Mov ax,0;
Jmp fimexp
Truexp:
Mov ax,1;
Fimexp:
Mov ds:[exp.end], ax
3
Mov ax, ds:[exps1.end]
Mov bx, ds:[exps2.end]
```

```
Cmp ax,bx
Jl truexp
Mov ax,0;
Jmp fimexp
Truexp:
Mov ax,1;
Fimexp:
Mov ds:[exp.end], ax
4
Mov ax, ds:[exps1.end]
Mov bx, ds:[exps2.end]
Cmp ax,bx
Jg truexp
Mov ax,0;
Jmp fimexp
Truexp:
Mov ax,1;
Fimexp:
Mov ds:[exp.end], ax
5
Mov ax, ds:[exps1.end]
Mov bx, ds:[exps2.end]
Cmp ax,bx
Jge truexp
Mov ax,0;
Jmp fimexp
Truexp:
Mov ax,1;
Fimexp:
Mov ds:[exp.end], ax
6
Mov ax, ds:[exps1.end]
Mov bx, ds:[exps2.end]
```

```
Cmp ax,bx
Jle truexp
Mov ax,0;
Jmp fimexp
Truexp:
Mov ax,1;
Fimexp:
Mov ds:[exp.end], ax
```

Função EXPS -> [+| -] T 1 {(+ 2| - 3| or4)T}

```
Mov ax, ds:[t1.end]
Mov bx, ds:[t2.end]
1
If (aux = ”-“){
        Neg ax
        Mov ds:[t1.end], ax
}
2
Add ax,bx
Mov ds:[exps.end], ax
3
Sub ax,bx
Mov ds:[exps.end], ax
4
Or ax,bx
Mov ds:[exps.end], ax
```

Função T -> F {(* 1 |and 2 | / 3 | % 4) F}

```
Mov ax, ds:[f1.end]
Mov bx, ds:[f2.end]
1
Imul bx
Mov ds:[t.end], ax
```

2

and ax,bx

Mov ds:[t.end], ax

3

Idiv bx

Mov ds:[t.end], ax

4

idiv bx

Mov ds:[t.end], dx

Função F -> N {(not) N 1}

Mov ax, ds:[n.end]

Mov ds:[F.end], ax

1

Mov ax, ds:[n.end]

Neg ax

Mov ds:[n.end], ax

Função N

1

Mov ax, ds:[id.end]

Mov ds:[n.end], ax

2

Mov ax, const

Mov ds:[n.end], ax

3

Mov ax, ds:[exp.end]

Mov ds:[n.end], ax