Trabalho de Compiladores – Linguagem L

Nome:
Breno Aroeira Cosenza
Guilherme De Andrade Moura
Rithie Natan Carvalhaes Prado

Analisador Léxico

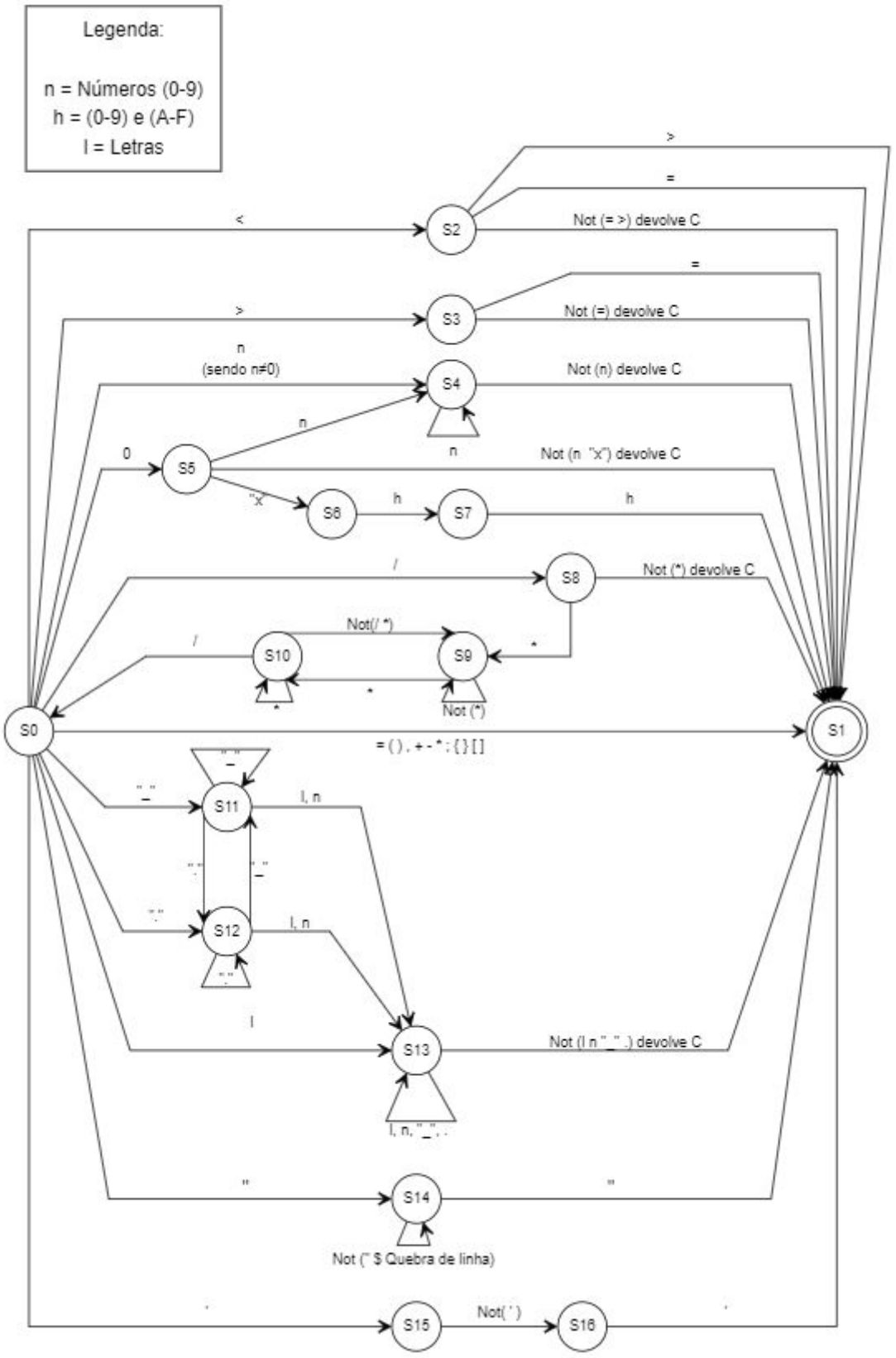
	Σ	**Lexema**
1	CONSTANTE	((0xhh) | n^+ | (‘z’))
2	VAR	var
3	INTEGER	integer
4	CHAR	char
5	FOR	for
6	IF	if
7	ELSE	else
8	AND	and
9	OR	or
10	NOT	not
11	IGUAL	=
12	TO	to
13	A_PARENTESES	(
14	F_PARENTESES	)
15	MENOR	<
16	MAIOR	>
17	DIFERENTE	<>
18	MAIOR IGUAL	>=
19	MENOR IGUAL	<=
20	VIRGULA	,
21	MAIS	+
22	MENOS	-
23	ASTERISCO	*
24	BARRA	/
25	PONTO_E_VIRGULA	;

26	A_CHAVE	{
27	F_CHAVE	}
28	THEN	then
29	READLN	readln
30	STEP	step
31	WRITE	write
32	WRITELN	writeln
33	PORCENTAGEM	%
34	A_COLCHETE	[
35	F_COLCHETE	]
36	DO	do
37	ID	(L(L | n | _ | .)*) | ((. | _)$^+$(L | n)(L | n | . |_)*)
38	CONST	const

Observações:

z - representa todos os caracteres permitidos

Autômato



Gramática LL(1)

S - > {DECLARACAO} {CMD}⁺ EOF
DECLARACAO -> Var {DEC_VAR}⁺ | DEC_CONST
DEC_VAR -> (Integer | Char) ID OPT_VAR {,ID OPT_VAR };
OPT_VAR -> [(“[“ CONSTANTE ”]” | = [-] CONSTANTE)]
DEC_CONST -> CONST ID = [-] CONSTANTE;
CMD -> ATR | REP | TESTE | NULO | RW
ATR -> ID [“[”EXP“]”] = EXP;
REP -> for ID ["["EXP"]"] = EXP to EXP [step EXP] do (CMD | "{" {CMD}+ "}")
TESTE -> if EXP then (CMD | “{“{CMD}⁺”}”) [Else (CMD | “{“{CMD}⁺”}”⁺)]
NULO -> ;
RW -> (readln”(“ID”)” | (write | writeln) “(“EXP {,EXP}”)");
EXP -> EXPS [(= | <> | < | > | <= | >=) EXPS]
EXPS -> [-] T {(+ | - | OR) T}
T -> F {(* | / | AND | %) F}
F -> NOT F | ID [“[“EXP”]”] | CONSTANTE | “(“EXP”)”

Esquema de Tradução

S - > {DECLARACAO} {CMD}⁺ EOF
DECLARACAO -> Var {DEC_VAR}⁺ | DEC_CONST
DEC_VAR -> (4)(Integer | Char) (5)ID OPT_VAR {,ID(6) OPT_VAR };
OPT_VAR -> (7)[(“[“ (8)CONSTANTE ”]” | = [-] (9)CONSTANTE)]
DEC_CONST -> CONST ID (1) = [-] (2)CONSTANTE(3);
CMD -> ATR | REP | TESTE | NULO | RW
ATR -> (10)ID [“[”EXP(11)“]”(12)] = EXP;
REP -> for (13)ID(61) ["["EXP(11)"]"(12)(62)] = EXP1(11)(63) to EXP2(11)(64) [step (14)CONSTANTE(65)] do (CMD | "{" {CMD}+ "}"(66))
TESTE -> if EXP(15)(58) then (CMD | “{“{CMD}⁺”}”) [Else (CMD | “{“{CMD}⁺”}”⁺)(59)](60)
NULO -> ;
RW -> (readln”(“ID(16)(53)”)” | (write(54) | writeln(55)) “(“EXP(17)(56) {,EXP(17)(57)}”)");
EXP -> EXPS(18)(46) [(=(19) | <>(47) | < (48) | >(49) | <=(50) | >=(51)) EXPS1(20)(52)]
EXPS -> (21)(45)[-(22)(46)] T (23)(47) {(+ (48)| - (49)| OR (24)) T1(25)(50)}
T -> F(26)(40) {(* (41)| /(42) | AND(27) | %(43)) F1(28)(44)}
F -> (35)(29)(NOT F1(30)(36) | (31)ID [“[“EXP (32)”]”(37)] | (33) CONSTANTE(38) | “(“EXP”)”)(34)

Legenda

1. if (tokenAtual.classe == Token.VAZIO){
   tokenAtual.classe = Token.CONST;
   }else{
   lexemaErro = analisadorLexico.lexema;
   MENSAGEM_ERRO = IDENTIFICADOR_JA_DECLARADO;
   }

2. if (tokenAtual.id == TokenID.MENOS){
   isNegativo = true;
   }

3. if (tokenAtual.tipo == STRING || (isNegativo && tokenAtual.tipo != INTEGER)){
   MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
   }

4. if(tokenAtual.id == TokenID.INTEGER){
   tipo = INTEGER;
   }else{
   tipo = CHAR;
   }

5. tokenAtual.tipo = tipo;
   if (tokenAtual.classe == Token.VAZIO){
   tokenAtual.classe = Token.VAR;
   }else{
   lexemaErro = analisadorLexico.lexema;
   MENSAGEM_ERRO = IDENTIFICADOR_JA_DECLARADO;
   }

6. tokenAtual.tipo = tipo;
   if (tokenAtual.classe == Token.VAZIO){
   tokenAtual.classe = Token.VAR;
   }else{
   lexemaErro = analisadorLexico.lexema;
   MENSAGEM_ERRO = IDENTIFICADOR_JA_DECLARADO;
   }

7. Token auxToken = tokenAnterior;

8.
```
if(tokenAtual.tipo != INTEGER){
    MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
}
if ((auxToken.tipo == CHAR && Integer.parseInt(tokenAtual.lexema) > 4000) || (auxToken.tipo
== INTEGER && Integer.parseInt(tokenAtual.lexema) > 8000))
    MENSAGEM_ERRO = TAMANHO_MAX_EXCEDIDO;
```

9.
```
if (tokenAtual.id == TokenID.MENOS){
    if (tokenAtual.tipo != INTEGER){
        MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
    }
}
if (auxToken.tipo != tokenAtual.tipo){
    MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
}
```

10.
```
if (tokenAtual.classe == Token.VAZIO){
    lexemaErro = analisadorLexico.lexema;
    MENSAGEM_ERRO = IDENTIFICADOR_NAO_DECLARADO;
}else if (tokenAtual.classe == Token.CONST){
    MENSAGEM_ERRO = CLASSE_INCOMPATIVEL;
    lexemaErro = analisadorLexico.lexema;
}
```

11.
```
if (exp() != INTEGER){
    MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
    errosSemanticos();
}
```

12.
```
if (auxToken.tamanho == 0){
    MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
    errosSemanticos();
}
```

13.
```
int tipoExp = exp();
if(tipoExp == STRING){
    if (auxToken.classe == Token.CONST || auxToken.tipo == INTEGER || auxToken.tamanho
        == 0 ){
        MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
    }
}else if (auxToken.tipo != tipoExp){
    MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
```

```
    }
    errosSemanticos();

14. if (tokenAtual.tipo != INTEGER){
        MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
    }

15. if (exp() != BOOLEAN){
        MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
        errosSemanticos();
    }

16. if (tokenAtual.classe == Token.CONST){
        lexemaErro = analisadorLexico.lexema;
        MENSAGEM_ERRO = CLASSE_INCOMPATIVEL;
    }else if (tokenAtual.classe != Token.VAR){
        lexemaErro = analisadorLexico.lexema;
        MENSAGEM_ERRO = IDENTIFICADOR_NAO_DECLARADO;
    }

17. if (exp() == BOOLEAN){
        MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
        errosSemanticos();
    }

18. int tipoExp = exps();
    boolean isEquals = false;

19. isEquals = true;

20. if (comparador){
        if (isEquals){
            if ((tipoExp == STRING && auxTipo != STRING) || tipoExp == INTEGER && auxTipo !=
INTEGER){
                MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
                errosSemanticos();
            }else if (tipoExp == CHAR || auxTipo == CHAR){
                MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
                errosSemanticos();
            }
        }else{
            if (tipoExp != INTEGER || auxTipo != INTEGER){
```

```
            MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
            errosSemanticos();
        }
    }
    tipoExp = BOOLEAN;
  }
  return tipoExp;

21. int tipoExps = NOTYPE;
  boolean isOr = false;

22. tipoExps = INTEGER;

23. int auxTipo = T();
  if (tipoExps == INTEGER){
    if (auxTipo != INTEGER){
      MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
    }
  }else{
    tipoExps = auxTipo;
  }
  return tipoExps;

24. isOr = true;

25. auxTipo = T();
  if (isOr){
    if (tipoExps != BOOLEAN || auxTipo != BOOLEAN){
      MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
      errosSemanticos();
    }
  }else if(tipoExps != INTEGER || auxTipo != INTEGER){
    MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
    errosSemanticos();
  }

26. int tipoT = F();
  boolean isAnd = false;

27. isAnd = true;

28. if (isAnd){
```

```
        if (tipoT != BOOLEAN || auxTipo != BOOLEAN){
            MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
            errosSemanticos();
        }
    }else if(tipoT != INTEGER || auxTipo != INTEGER){
        MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
        errosSemanticos();
    }
    return tipoT;

29. int tipoF = NOTYPE;

30. tipoF = BOOLEAN;
    if (F() != BOOLEAN){
        MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
        errosSemanticos();
    }

31. tipoF = tokenAtual.tipo;
    boolean isVetor = false;
    if (tokenAtual.classe == Token.VAZIO){
        MENSAGEM_ERRO = IDENTIFICADOR_NAO_DECLARADO;
        lexemaErro = analisadorLexico.lexema;
    }
    Token auxToken = tokenAtual;

32. isVetor = true;
    if (auxToken.classe == Token.CONST){
        lexemaErro = analisadorLexico.lexema;
        MENSAGEM_ERRO = CLASSE_INCOMPATIVEL;
    }else if (exp() != INTEGER){
        MENSAGEM_ERRO = TIPOS_INCOMPATIVEIS;
        errosSemanticos();
    }

33. tipoF = tokenAtual.tipo;

34. return tipoF;

35. F.end = NTEMP
36. mov AX, DS:[F1.end]
      neg AX
```

```
    add AX, 1
    mov F.end, AX
37. mov AX, DS:[EXP.end]
    if (EXP.tipo == int){
        add AX, DS:[EXP.end]
    }
    add AX, DS:[ID.end]
    mov DS:[F.end], AX

38. mov AX, CONSTANTE.lex
    mov DS: [F.end], AX

39. F.tipo = EXP.tipo
    F.end = EXP.end

40. T.tipo = F.tipo
    T.end = F.end

41. isMult = true

42. isDiv = true;

43. isResto = true;

44. mov AX, DS:[F.end]
    mov BX, DS:[F1.end]
    if (isMult){
        imul BX
    }else if (isDiv){
        idiv BX
    }else if (AND){
        and AX,BX
    }else{
        idiv BX
        mov AX, DX
    }
    mov DS:[F.end], AX

45. boolean isNeg = false;
    boolean isSom = false;
    boolean isSub = false;
46. isNeg = true
```

```
47. EXPS.tipo = T.tipo
    EXPS.end = T.end
    mov AX, DS:[EXPS.end]
    if (isNeg){ neg AX }
48. isSom = true;
49. isSub = true;
50. mov BX, DS:[T1.end]
    if (isSom){
        add AX,BX
    }else if (isSub){
        neg BX
        add AX,BX
    }else if (isOr){
        or AX,BX
    }
    mov DS:[EXPS.end], AX

46. EXP.tipo = EXPS.tipo
    EXP.end = EXPS.end
    boolean isDif = false;
    boolean isMin = false;
    boolean isMaj = false;
    boolean isMinE = false;
    boolean isMajE = false;

47. isDif  = true;
48. isMin  = true;
49. isMaj  = true;
50. isMinE  = true;
51. isMajE  = true;
52. mov AX, DS:[EXP.end]
    mov BX, DS:[EXPS1.end]
    cmp AX, BX
    RotF = NOVOROT
    if (isDif){
        jne Rotf;
    }else if (isMin){
        jl RotF;
    }else if (isMaj){
        jg Rotf;
    }else if (isMinE){
        jle  Rotf;
```

```
    }else if (isMajE){
        jge  Rotf;
    }else{
        je Rotf;
    }
    mov AX, 0
    RotF2 = NOVOROT
    jmp RotF2
    RotF: mov AX, 1
    RotF2: mov DS:[EXP.end], AX
```

53.
54.
55.
56.
57.

58.
```
boolean isElse = false;
mov AX, DS:[EXP.end]
RotF = NOVOROT
cmp AX, 1
jne RotF
```

59.
```
boolean isElse = true;
```

60.
```
if (isElse){
        RotFim = NOVOROT;
        jmp RotFim
        RotF:
        CMD(); (Se tiver chaves vão ser “n” comandos, caso contrário apenas um)
        RotFim:
   }
```

61.
```
mov AX, DS:[ID.end]
mov CX, 1
```

62.
```
add DS:[EXP.end
if (EXP.tipo == INTEGER){
    + EXP.end
}
]
```

63.
```
mov AX, DS:[EXP1.end]
```

64. mov DX, DS:[EXP2.end]
65. mov CX, CONSTANTE.lexema
66. RotF = NOVOROT
    RotFim = NOVOROT
    RotF: cmp AX, DX
    je RotFim
    CMD(); (Se tiver chaves vão ser “n” comandos, caso contrário apenas um)
    add AX, CX
    jmp RotF
    RotFim: