Instituto Tecnológico de Aeronáutica

Curso de Engenharia da Computação

Disciplina CES-41: Compiladores

Relatório do Laboratório 2

Vitor Pimenta dos Reis Arruda

Professor Fábio Carneiro Mokarzel

São José dos Campos

12/03/2019

Sumário

1 Introdução

Este trabalho intenciona a escrita e o teste de um analisador léxico para a linguagem de programação COMP-ITA 2019 [1]. Implementa-se uma solução que processa programas escritos na tal linguagem e identifica seus **tokens**, relatando informações associadas a eles; também se testa a solução proposta a partir de amostras de código escolhidas para evidenciar o correto funcionamento do analisador léxico em variados contextos.

Para o desenvolvimento, usou-se a linguagem de programação **javascript** com o analisador léxico **Moo** [3]. Nota-se que o roteiro deste laboratório pede explicitamente ([2]) o uso da ferramenta **Flex**, que não foi utilizada, como já explicado.

As seções seguintes se estruturam de maneira que a primeira explica o funcionamento básico do **Moo** e as seguintes projetam o analisador específico desejado (para a linguagem COMP-ITA 2019).

2 Sintaxe da biblioteca Moo

Considere , como exemplo, o problema de detectar cadeias formadas por caracteres 0 e 1 nas quais a quantidade de 0 seja par ou a quantidade de 1 seja par. A Codificação 1 é sua solução em **Moo**:

```
1  const lexer1 = moo.compile({
2     valido: /(?:(?:0*10*10*)+|(?:1*01*01*)+|0+|1+)(?![^])/,
3     invalido: { match: /[^]+/, lineBreaks: true }
4  })
```

Codificação 1: Sintaxe Moo para detectar cadeias com quantidade par ou de 1 ou de 0

A função **moo.compile()** toma como argumento um objeto (delimitado por **{** na linha 1 e **}** na linha 4). Na linguagem **javascript**, um objeto é um dicionário, isto é, um conjunto de pares chave → valor. No exemplo, as chaves são **valido** e **invalido**, e o valor de cada uma é o que segue o sinal de dois-pontos (**:**).

Para a chave **valido**, forneceu-se uma expressão regular (delimitada pelos sinais / na linha 2). Para a biblioteca **Moo**, isso significa que, ao realizar posterior análise léxica de uma cadeia de caracteres, qualquer sequência aceita pela expressão regular será retornada como um **token**, que é um objeto

cujas chaves informam sobre a localização em que foi descoberto dentro da sequência de entrada (como linha e coluna), o tipo do **token** (que seria **valido** no caso do exemplo), seu texto (cópia literal da parte capturada da cadeia de entrada) e outros.

Para ilustrar, considere a Figura 1, obtida ao acionar **lexer1** sobre a cadeia **0011000** (que, nota-se, tem quantidade par de dígitos 1, como será discutido adiante).

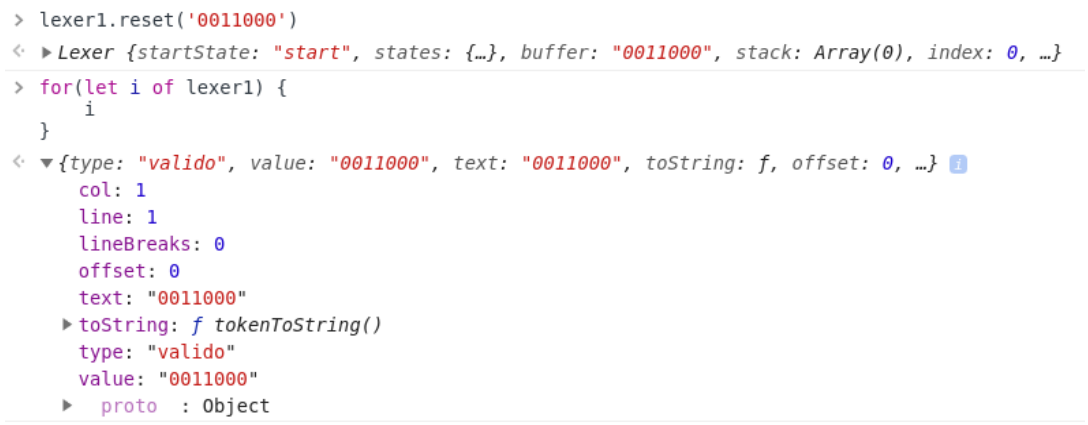

Figura 1: Saída de **lexer1** *sobre cadeia* **0011000**, *evidenciando a estrutura do objeto* **token** *retornado*

Quanto à outra chave (**invalido**) do objeto fornecido ao **Moo**, ela caracteriza outro tipo de **token**. Para preenchê-la, não se forneceu diretamente uma expressão regular (como no caso anterior), mas outro objeto contendo chaves **match** e **lineBreaks**. O efeito, porém, é o mesmo do caso anterior, e a necessidade de se fornecer um objeto em vez de uma expressão regular é meramente técnica: a biblioteca **Moo** exige que se explicite quando a expressão regular fornecida puder aceitar caracteres de mudança de linha (**\n**), daí a necessidade de especificar **lineBreaks:true**. Nota-se, também, que ***Moo*** não aceita expressões regulares que aceitem a cadeia vazia.

Por último, antes de discorrer sobre a lógica do problema em si, dois detalhes de sintaxe das expressões regulares: Os grupos envolvidos pelos sinais **(?:** e **)** são chamados **grupos não capturantes** e possuem o mesmo efeito da parentisação usual (isto é, sem ***?:***) no que concerne à aceitação de cadeias, enquanto os grupos envolvidos por ***(?!*** e ***)*** são chamados **lookahead negativos**, e servem para especificar que a expressão regular anterior a eles só deve aceitar certa cadeia se esta **não for** sucedida pelo que estiver dentro do **lookahead negativo**. Em relação a outros aspectos do formalismo de expressões regulares, por serem similares aos do **Flex**, são assumidos como conhecidos pelo leitor.

Finalmente, explica-se o motivo de a expressão regular fornecida para **valido** resolver ao problema proposto:

- A porção **(?:0*10*10*)+** identifica cadeias com quantidade par *não-nula* de dígitos 1
- A porção **(?:1*01*01*)+** identifica cadeias com quantidade par *não-nula* de dígitos 0
- A porção **0+** identifica cadeias sem senhum dígito 1 (afinal, 0 é par)
- A porção **1+** identifica cadeias sem senhum dígito 0 (afinal, 0 é par)
- O **lookahead negativo (?![^])** determina que, depois da cadeia aceita por qualquer dos pontos anteriores, não deve aparecer nenhum outro caractere (**[^]** é uma expressão regular que aceita qualquer caractere único, inclusive **\n**).

Reconhece-se que, a rigor, a cadeia vazia deveria ser aceita pela solução proposta, pois é solução válida do problema. Porém, como **Moo** não permite o uso de expressões regulares que aceitem a cadeia vazia, não é possível corrigir esse erro unicamente com a biblioteca. Portanto, para que a solução seja completamente correta (avaliando corretamente a cadeia vazia), é necessário fazer essa verificação "por fora" da biblioteca, como na Codificação 2:

```
1    export function predicate1(inputstring: string) {
2        if (inputstring === '') {
3            return true
4        }
5        lexer1.reset(inputstring)
6        return lexer1.next().type === 'valido'
7    }
```

Codificação 2: Função-solução ao problema-exemplo. Dada uma entrada **inputstring**, *ela retorna o valor lógico* **true** *se, e somente se, a entrada for solução do problema*

3 Analisador léxico para COMP-ITA 2019

A subseção seguinte resume a especificação léxica da linguagem de interesse, enquanto as outras subseções apresentam o analisador léxico projetado para ela.

3.1 Especificação formal

3.1.1 Palavras reservadas (todas em letras minúsculas)

call	**char**	**do**	**else**	**false**	**float**
for	**functions**	**global**	**if**	**int**	**local**
logic	**main**	**program**	**read**	**return**	**statements**
true	**void**	**while**	**write**		

3.1.2 Sintaxes

Descrição	Tipo	Sintaxe
Identificador	**ID**	**Letra (Letra | Dígito)***
Constante inteira	**INTCT**	**(Dígito) +**
Constante caractere	**CHARCT**	**'(\)? Caractere'**
Constante real	**FLOATCT**	**(Dígito)+ . (Dígito)* ((E | e) (+ | -)? (Dígito) +)?**
Cadeia de caracteres	**STRING**	**" ((\)? Caractere)* "**

3.1.3 Operadores

Classe	Tipo	Operadores
Or lógico	**OR**	**||**
And lógico	**AND**	**&&**
Not lógico	**NOT**	**!**
Relacionais	**RELOP**	**< <= > >= = !=**
Aditivos	**ADOP**	**+ -**
Multiplicativos	**MULTOP**	*** / %**
Negadores	**NEG**	**~**

Obs.: Os operadores de tipos **OR**, **AND**, **NOT** e **NEG** não precisam de atributos.

3.1.4 Separadores e outros (não precisam de atributos)

Descrição	Símbolo	Tipo	Descrição	Símbolo	Tipo
Atribuição	**<-**	**ASSIGN**	**Abre-chave**	**{**	**OPBRACE**
Abre-parêntesis	**(**	**OPPAR**	**Fecha-chave**	**}**	**CLBRACE**
Fecha-parêntesis	**)**	**CLPAR**	**Ponto e vírgula**	**;**	**SCOLON**
Abre-colchete	**[**	**OPBRAK**	**Vírgula**	**,**	**COMMA**
Fecha-colchete	**]**	**CLBRAK**	**Dois pontos**	**:**	**COLON**

3.1.5 Comentários nos programas:

Tudo o que estiver entre **/*** e ***/**

3.2 Solução proposta

Considere o analisador definido pela Codificação 3:

```
1  const letra_regexp = '[a-zA-Z]'
2  const digit_regexp = '[0-9]'
3  const carac1_regexp = '(?:\\\\.|[^\\\\\'\n])'
4  const carac2_regexp = '(?:\\\\.|[^\\\\"\n])'
5
6  const lexer = moo.compile({
7      COMENTARIO: {
8          match: /\/\*[^]*?\*\//,
9          lineBreaks: true,
10         value: empty_string
11     },
12     ID: {
13         match: new RegExp(`${letra_regexp}(?:${letra_regexp}|${digit_regexp})*`),
14         type: moo.keywords({
15             CALL: 'call',
16             FOR: 'for',
```

```
17            LOGIC: 'logic',
18            TRUE: 'true',
19            CHAR: 'char',
20            FUNCTIONS: 'functions',
21            MAIN: 'main',
22            VOID: 'void',
23            DO: 'do',
24            GLOBAL: 'global',
25            PROGRAM: 'program',
26            WHILE: 'while',
27            ELSE: 'else',
28            IF: 'if',
29            READ: 'read',
30            WRITE: 'write',
31            FALSE: 'false',
32            INT: 'int',
33            RETURN: 'return',
34            FLOAT: 'float',
35            LOCAL: 'local',
36            STATEMENTS: 'statements'
37        })
38    },
39    FLOATCT: {
40        match: new RegExp(`${digit_regexp}+\\.${digit_regexp}*(?:(?:E|e)(?:\\+|-)?${digit_regexp}+)?`),
41        value: text => (parseFloat(text) as unknown) as string
42    },
43    INTCT: {
44        match: new RegExp(`${digit_regexp}+`),
45        value: text => (parseInt(text, 10) as unknown) as string
46    },
47    CHARCT: {
48        match: new RegExp(`'${carac1_regexp}'`),
49        value: text => text.slice(1, -1) // descarta os sinais de apóstrofe (primeira e última posições)
50    },
51    STRING: {
52        match: new RegExp(`"(?:${carac2_regexp})*?"`),
53        value: text => text.slice(1, -1) // descarta os sinais de aspas (primeira e última posições)
54    },
55    OR: {
56        match: /\|\|/,
57        value: empty_string
58    },
59    AND: {
60        match: /&&/,
61        value: empty_string
62    },
63    RELOP: {
64        match: /<(?!=|-)|<=|>(?!=)|>=|=|!=/,
65        value: text => ({
66            '<': 'LT',
```

```
67            '<=': 'LE',
68            '>': 'GT',
69            '>=': 'GE',
70            '=': 'EQ',
71            '!=': 'NE'
72        }[text])
73    },
74    NOT: {
75        match: /!/,
76        value: empty_string
77    },
78    ADOP: {
79        match: /\+|-/,
80        value: text => ({
81            '+': 'MAIS',
82            '-': 'MENOS'
83        }[text])
84    },
85    MULTOP: {
86        match: /\*|\/|%/,
87        value: text => ({
88            '*': 'VEZES',
89            '/': 'DIV',
90            '%': 'RESTO'
91        }[text])
92    },
93    NEG: {
94        match: /~/,
95        value: empty_string
96    },
97    ASSIGN: {
98        match: /<-/,
99        value: empty_string
100   },
101   OPPAR: {
102       match: /\(/,
103       value: empty_string
104   },
105   CLPAR: {
106       match: /\)/,
107       value: empty_string
108   },
109   OPBRAK: {
110       match: /\[/,
111       value: empty_string
112   },
113   CLBRAK: {
114       match: /\]/,
115       value: empty_string
116   },
117   OPBRACE: {
118       match: /\{/,
```

```
119         value: empty_string
120     },
121     CLBRACE: {
122         match: /\}/,
123         value: empty_string
124     },
125     SCOLON: {
126         match: /;/,
127         value: empty_string
128     },
129     COMMA: {
130         match: /,/,
131         value: empty_string
132     },
133     COLON: {
134         match: /:/,
135         value: empty_string
136     },
137     WHITESPACE: {
138         match: /\s+/,
139         lineBreaks: true,
140         value: empty_string
141     },
142     INVALIDO: /./
143 })
```

Codificação 3: Sintaxe Moo para analisador léxico da linguagem COMP-ITA 2019

Como **Moo** favorece as primeiras definições regulares introduzidas, foi necessário definir **COMENTARIO** antes dos outros **tokens** da linguagem – caso contrário, a / que inicia um comentário poderia ser interpretada como **MULTOP**, por exemplo. Além disso, foi necessário usar (na expressão regular) o operador ***?**, que detecta zero ou mais ocorrências do grupo anterior de maneira **não-gulosa**: isso significa que uma cadeia como **/* *este é um comentário* */ /* aqui é outro comentário */** não é interpretada como um único comentário (começando no primeiro **/*** e terminando no último ***/**), mas sim como dois.

Em seguida, transcreveu-se a definição de um **ID** (conforme a especificação) e já se empregou um facilitador da biblioteca: **moo.keywords**, que realiza automaticamente a desambiguação entre identificador e palavra reservada – por exemplo, esse facilitador corretamente interpreta **do** como palavra reservada e **doing** como identificador.

Em relação às constantes numéricas, mais uma vez a ordem foi crucial: **FLOATCT** requereu ser definida antes de **INTCT**, porque **tokens** com a segunda classificação podem ser prefixos de **tokens** com a primeira classificação. Caso a ordem das definições fosse invertida, o texto **12.3** seria interpretado como um **INTCT=12**, um caractere inválido **.** e um outro **INTCT=3**, quando o desejado é **FLOATCT=12.3**. Nas definições desses dois **tokens**, observe a chave **value** do dicionário fornecido ao **Moo**: essa chave armazena uma função aplicada a todo **token** capturado, e é responsável por preencher o **atributo** do **token** com o valor *numérico* da constante.

Adiante, definem-se os **tokens** de texto: **CHARCT** e **STRING**. O primeiro é trivial, mas o segundo tem o mesmo potencial problema de **COMENTARIO**: caso não fosse usado novamente o operador

***?**, uma cadeia como **“aqui é uma string” “aqui é outra string”** seria erroneamente interpretada como uma única **STRING** (começando na primeira **“** e terminando na última **”**), quando o correto é identificá-la como contendo duas **STRING**. Mais uma vez, a chave **value** é usada, agora para retirar os delimitadores das constantes ao armazenar seus conteúdos (isto é, as **‘** no caso de **CHARCT** e **“** no caso de **STRING**).

Quanto aos operadores e separadores definidos na sequência, dispensam comentário em sua maioria, a não ser pelo fato de alguns gerarem o mesmo tipo de problema já mencionado para constantes numéricas: quando um operador é prefixo de outro, a ordenação das definições é crucial. Por exemplo, o operador **!** (**NOT**) é prefixo de **!=** (**NE**), donde o primeiro teve que ser definido após o segundo. Além disso, observe a expressão regular para **RELOP**, em que aparece o operador de **lookahead negativo** (identificado em **(?!=|-)** e **(?!=)**), usado também para evitar o conflito de prefixos (agora envolvendo os sinais de “maior” e “menor”).

As últimas definições são para **WHITESPACE** e **INVALIDO**. Quanto ao primeiro, ele é identificado, mas também descartado logo em seguida (como para **COMENTARIO**); já o segundo captura todo caractere que apareça na cadeia de entrada e não se encaixe em nenhuma das categorias precedentes.

Finalmente, o analisador produzido pela biblioteca **Moo** foi envolvido pela função **lex** (Codificação 4), que aceita uma cadeia de entrada e produz uma lista de **tokens**, cada um dos quais possui **Texto**, **Tipo** e **Atributo** (apesar de alguns terem atributo vazio):

```
1  export function lex(inputstring: string) {
2      const tokens = []
3      lexer.reset(inputstring)
4      let token
5      while ((token = lexer.next()) !== undefined) {
6          if (token.type === 'WHITESPACE' || token.type === 'COMENTARIO') {
7              continue
8          }
9          tokens.push({
10             Texto: token.text,
11             Tipo: token.type,
12             Atributo: token.value
13         })
14     }
15     return tokens
16 }
```

Codificação 4: Função final para análise léxica de COMP-ITA 2019

3.3 Testes realizados

Primeiramente, checou-se a funcionalidade geral do analisador sobre um arquivo de código fornecido pelo professor e visto na Codificação 5, o qual faz uso de quase todos os **tokens** da linguagem (mas não todos !):

```
1  /*  Programa para contar as ocorrencias das palavras de um texto */
2
3  program AnaliseDeTexto {
```

```
4
5   /*  Variaveis globais  */
6
7   global:
8      char nomes[50,10], palavra[10];
9      int ntab, nocorr[50];
10        char c; logic fim;
11
12  functions:
13
14  /*  Funcao para procurar uma palavra na tabela de palavras  */
15
16  int Procura () {
17
18  local:
19     int i, inf, sup, med, posic, compara;
20        logic achou, fimteste;
21  statements:
22     achou <- false; inf <- 1; sup <- ntab;
23     while (!achou && sup >= inf) {
24        med <- (inf + sup) / 2;
25        compara <- 0; fimteste <- false;
26        for (i <- 0; !fimteste && compara = 0; i <- i+1) {
27              if (palavra[i] < nomes[med,i])
28                compara <- ~1;
29              else if (palavra[i] > nomes[med,i])
30                compara <- 1;
31              if (palavra[i] = '\0' || nomes[med,i] = '\0')
32                fimteste <- true;
33        }
34        if (compara = 0)
35           achou <- true;
36        else if (compara < 0)
37           sup <- med - 1;
38        else inf <- med + 1;
39     }
40     if (achou) posic <- med;
41     else posic <- ~inf;
42     return posic;
43
44  } /* Fim da funcao Procura */
45
46  /*  Funcao para inserir uma palavra na tabela de palavras  */
47
48  void Inserir (int posic) {
49
50  local:
51     int i, j; logic fim;
52  statements:
53     ntab <- ntab + 1;
54     for (i <- ntab; i >= posic+1; i <- i-1) {
55        fim <- false;
```

```
56            for (j <- 0; !fim; j <- j+1) {
57               nomes[i,j] <- nomes[i-1,j];
58               if (nomes[i,j] = '\0') fim <- true;
59            }
60           nocorr[i] <- nocorr[i-1];
61        }
62            fim <- false;
63            for (j <- 0; !fim; j <- j+1) {
64               nomes[posic,j] <- palavra[j];
65               if (palavra[j] = '\0') fim <- true;
66            }
67        nocorr[posic] <- 1;
68
69    } /* Fim da funcao Inserir */
70
71    /*  Funcao para escrever a tabela de palavras  */
72
73    void ExibirTabela () {
74
75    local:
76        int i; logic fim;
77    statements:
78        write ("          ", "Palavra              ",
79                              "   Num. de ocorr.");
80        for (i <- 1; i <= 50; i <- i+1) write ("-");
81        for (i <- 1; i <= ntab; i <- i+1) {
82            write ("\n          "); fim <- false;
83            for (j <- 0; !fim; j <- j+1) {
84               if (nomes[i,j] = '\0') fim <- true;
85               else write (nomes[i,j]);
86            }
87            write (" | ", nocorr[i]);
88        }
89
90    } /* Fim da funcao ExibirTabela */
91
92
93    /*  Modulo principal  */
94
95    main {
96
97    local:
98        int i, posic;
99            char c; logic fim;
100   statements:
101       ntab <- 0;
102       write ("Nova palavra? (s/n): ");
103       read (c);
104       while (c = 's' || c = 'S') {
105               write ("\nDigite a palavra: ");
106               fim <- false;
107           for (i <- 0; !fim; i <- i+1) {
```

```
108                 read (palavra[i]);
109                 if (palavra[i] = '\n') {
110                     fim <- true;
111                     palavra[i] <- '\0';
112                 }
113             }
114         posic <- Procura ();
115         if (posic > 0)
116             nocorr[posic] <- nocorr[posic] + 1;
117         else
118             call Inserir (~posic, i);
119             write ("\n\nNova palavra? (s/n): ");
120             read (c);
121     }
122     call ExibirTabela ();
123
124 } /* Fim da funcao main */
125
126 } /* Fim do programa AnaliseDeTexto */
```

Codificação 5: Programa-exemplo fornecido pelo professor

Quando submetido ao analisador implementado, o resultado (Apêndice A) é gerado de acordo com o esperado, como se verifica por inspeção.

Por último, gerou-se um arquivo-código com o único propósito de exercitar casos problemáticos (como os ilustrados na seção 3.2). O código tem vários caracteres inválidos e cada linha sua descreve uma situação que poderia gerar erro no analisador (Codificação 6):

```
1  /* comentario, jogue fora *** */ /* este tambem */
2  "abc""def" sao 2 strings;
3  2 % 3 / 4 + 5 - 6 * 7 sao operacoes;
4  '\' '\" ' ' sao /* nem todos */  CHARCT;
5  "ab\"c" nao eh possivel escape de aspas;
6  nada de ácentos;
7  12.3 eh float, nao int;
8  < eh menor, <= eh menor ou igual;
9  > eh maior, >= eh maior ou igual;
10 em ID, soh letras_mais_nada;
11 != eh diferente, ! eh negacao, = eh igualdade;
12 CHARCT '
13 ' nao pode ser multi-linhas;
14 strings "
15                 " nao podem ser multi-linhas tambem;
16 do eh keyword; doing, nao;
17 '\' ; '"; nao sao CHARCT, mas '\" eh
```

Codificação 6: Código para exercício de casos problemáticos ao analisador

Mais uma vez, os resultados gerados (Apêndice B) sugerem correto funcionamento do analisador projetado.

4 Experimentação do código

Para testar o código usado, basta abrir o arquivo **index.html** fornecido junto com este laboratório em um navegador e seguir as instruções da página.

5 Referências

[1] Mokarzel, F. C. 2º **Linguagem COMP-ITA 2019**. Especificação formal de uma linguagem de programação utilizada na disciplina CES-41 do Instituto Tecnológico de Aeronáutica, 2019.

[2] Mokarzel, F. C. 2º **Laboratório de CES-41/2019**. Roteiro do laboratório 2 da disciplina CES-41 do Instituto Tecnológico de Aeronáutica, 2019.

[3] Radvan, T. **Moo**. Analisador léxico escrito em *javascript*. Repositório em: <https://github.com/no-context/moo>. Acesso em 04/03/2019.

APÊNDICE A – ANÁLISE LÉXICA DO PROGRAMA EM COMP-ITA 2019 FORNECIDO PELO PROFESSOR

Texto	**Tipo**	**Atributo**
program	PROGRAM	program
AnaliseDeTexto	ID	AnaliseDeTexto
{	OPBRACE	
global	GLOBAL	global
:	COLON	
char	CHAR	char
nomes	ID	nomes
[	OPBRAK	
50	INTCNT	50
,	COMMA	
10	INTCNT	10
]	CLBRAK	
,	COMMA	
palavra	ID	palavra
[	OPBRAK	
10	INTCNT	10
]	CLBRAK	
;	SCOLON	
int	INT	int
ntab	ID	ntab
,	COMMA	
nocorr	ID	nocorr

[	OPBRAK	
50	INTCNT	50
]	CLBRAK	
;	SCOLON	
char	CHAR	char
c	ID	c
;	SCOLON	
logic	LOGIC	logic
fim	ID	fim
;	SCOLON	
functions	FUNCTIONS	functions
:	COLON	
int	INT	int
Procura	ID	Procura
(	OPPAR	
)	CLPAR	
{	OPBRACE	
local	LOCAL	local
:	COLON	
int	INT	int
i	ID	i
,	COMMA	
inf	ID	inf
,	COMMA	

sup	ID	sup
,	COMMA	
med	ID	med
,	COMMA	
posic	ID	posic
,	COMMA	
compara	ID	compara
;	SCOLON	
logic	LOGIC	logic
achou	ID	achou
,	COMMA	
fimteste	ID	fimteste
;	SCOLON	
statements	STATEMENTS	statements
:	COLON	
achou	ID	achou
<-	ASSIGN	
false	FALSE	false
;	SCOLON	
inf	ID	inf
<-	ASSIGN	
1	INTCNT	1
;	SCOLON	
sup	ID	sup

<-	ASSIGN	
ntab	ID	ntab
;	SCOLON	
while	WHILE	while
(	OPPAR	
!	NOT	
achou	ID	achou
&&	AND	
sup	ID	sup
>=	RELOP	GE
inf	ID	inf
)	CLPAR	
{	OPBRACE	
med	ID	med
<-	ASSIGN	
(	OPPAR	
inf	ID	inf
+	ADOP	MAIS
sup	ID	sup
)	CLPAR	
/	MULTOP	DIV
2	INTCNT	2
;	SCOLON	
compara	ID	compara
<-	ASSIGN	

0	INTCNT	0
;	SCOLON	
fimteste	ID	fimteste
<-	ASSIGN	
false	FALSE	false
;	SCOLON	
for	FOR	for
(	OPPAR	
i	ID	i
<-	ASSIGN	
0	INTCNT	0
;	SCOLON	
!	NOT	
fimteste	ID	fimteste
&&	AND	
compara	ID	compara
=	RELOP	EQ
0	INTCNT	0
;	SCOLON	
i	ID	i
<-	ASSIGN	
i	ID	i
+	ADOP	MAIS
1	INTCNT	1
)	CLPAR	

{	OPBRACE	
if	IF	if
(	OPPAR	
palavra	ID	palavra
[	OPBRAK	
i	ID	i
]	CLBRAK	
<	RELOP	LT
nomes	ID	nomes
[	OPBRAK	
med	ID	med
,	COMMA	
i	ID	i
]	CLBRAK	
)	CLPAR	
compara	ID	compara
<-	ASSIGN	
~	NEG	
1	INTCNT	1
;	SCOLON	
else	ELSE	else
if	IF	if
(	OPPAR	
palavra	ID	palavra
[	OPBRAK	

i	ID	i
]	CLBRAK	
>	RELOP	GT
nomes	ID	nomes
[	OPBRAK	
med	ID	med
,	COMMA	
i	ID	i
]	CLBRAK	
)	CLPAR	
compara	ID	compara
<-	ASSIGN	
1	INTCNT	1
;	SCOLON	
if	IF	if
(	OPPAR	
palavra	ID	palavra
[	OPBRAK	
i	ID	i
]	CLBRAK	
=	RELOP	EQ
'\0'	CHARCT	\0
||	OR	
nomes	ID	nomes
[	OPBRAK	

med	ID	med
,	COMMA	
i	ID	i
]	CLBRAK	
=	RELOP	EQ
'\0'	CHARCT	\0
)	CLPAR	
fimteste	ID	fimteste
<-	ASSIGN	
true	TRUE	true
;	SCOLON	
}	CLBRACE	
if	IF	if
(	OPPAR	
compara	ID	compara
=	RELOP	EQ
0	INTCNT	0
)	CLPAR	
achou	ID	achou
<-	ASSIGN	
true	TRUE	true
;	SCOLON	
else	ELSE	else
if	IF	if
(	OPPAR	

compara	ID	compara
<	RELOP	LT
0	INTCNT	0
)	CLPAR	
sup	ID	sup
<-	ASSIGN	
med	ID	med
-	ADOP	MENOS
1	INTCNT	1
;	SCOLON	
else	ELSE	else
inf	ID	inf
<-	ASSIGN	
med	ID	med
+	ADOP	MAIS
1	INTCNT	1
;	SCOLON	
}	CLBRACE	
if	IF	if
(	OPPAR	
achou	ID	achou
)	CLPAR	
posic	ID	posic
<-	ASSIGN	
med	ID	med

;	SCOLON	
else	ELSE	else
posic	ID	posic
<-	ASSIGN	
~	NEG	
inf	ID	inf
;	SCOLON	
return	RETURN	return
posic	ID	posic
;	SCOLON	
}	CLBRACE	
void	VOID	void
Inserir	ID	Inserir
(	OPPAR	
int	INT	int
posic	ID	posic
)	CLPAR	
{	OPBRACE	
local	LOCAL	local
:	COLON	
int	INT	int
i	ID	i
,	COMMA	
j	ID	j
;	SCOLON	

logic	LOGIC	logic
fim	ID	fim
;	SCOLON	
statements	STATEMENTS	statements
:	COLON	
ntab	ID	ntab
<-	ASSIGN	
ntab	ID	ntab
+	ADOP	MAIS
1	INTCNT	1
;	SCOLON	
for	FOR	for
(	OPPAR	
i	ID	i
<-	ASSIGN	
ntab	ID	ntab
;	SCOLON	
i	ID	i
>=	RELOP	GE
posic	ID	posic
+	ADOP	MAIS
1	INTCNT	1
;	SCOLON	
i	ID	i

<-	ASSIGN	
i	ID	i
-	ADOP	MENOS
1	INTCNT	1
)	CLPAR	
{	OPBRACE	
fim	ID	fim
<-	ASSIGN	
false	FALSE	false
;	SCOLON	
for	FOR	for
(	OPPAR	
j	ID	j
<-	ASSIGN	
0	INTCNT	0
;	SCOLON	
!	NOT	
fim	ID	fim
;	SCOLON	
j	ID	j
<-	ASSIGN	
j	ID	j
+	ADOP	MAIS
1	INTCNT	1
)	CLPAR	

{	OPBRACE	
nomes	ID	nomes
[	OPBRAK	
i	ID	i
,	COMMA	
j	ID	j
]	CLBRAK	
<-	ASSIGN	
nomes	ID	nomes
[	OPBRAK	
i	ID	i
-	ADOP	MENOS
1	INTCNT	1
,	COMMA	
j	ID	j
]	CLBRAK	
;	SCOLON	
if	IF	if
(	OPPAR	
nomes	ID	nomes
[	OPBRAK	
i	ID	i
,	COMMA	
j	ID	j
]	CLBRAK	

=	RELOP	EQ
'\0'	CHARCT	\0
)	CLPAR	
fim	ID	fim
<-	ASSIGN	
true	TRUE	true
;	SCOLON	
}	CLBRACE	
nocorr	ID	nocorr
[	OPBRAK	
i	ID	i
]	CLBRAK	
<-	ASSIGN	
nocorr	ID	nocorr
[	OPBRAK	
i	ID	i
-	ADOP	MENOS
1	INTCNT	1
]	CLBRAK	
;	SCOLON	
}	CLBRACE	
fim	ID	fim
<-	ASSIGN	
false	FALSE	false
;	SCOLON	

for	FOR	for
(	OPPAR	
j	ID	j
<-	ASSIGN	
0	INTCNT	0
;	SCOLON	
!	NOT	
fim	ID	fim
;	SCOLON	
j	ID	j
<-	ASSIGN	
j	ID	j
+	ADOP	MAIS
1	INTCNT	1
)	CLPAR	
{	OPBRACE	
nomes	ID	nomes
[	OPBRAK	
posic	ID	posic
,	COMMA	
j	ID	j
]	CLBRAK	
<-	ASSIGN	
palavra	ID	palavra
[	OPBRAK	

j	ID	j
]	CLBRAK	
;	SCOLON	
if	IF	if
(	OPPAR	
palavra	ID	palavra
[	OPBRAK	
j	ID	j
]	CLBRAK	
=	RELOP	EQ
'\0'	CHARCT	\0
)	CLPAR	
fim	ID	fim
<-	ASSIGN	
true	TRUE	true
;	SCOLON	
}	CLBRACE	
nocorr	ID	nocorr
[	OPBRAK	
posic	ID	posic
]	CLBRAK	
<-	ASSIGN	
1	INTCNT	1
;	SCOLON	
}	CLBRACE	

void	VOID	void
ExibirTabela	ID	ExibirTabela
(	OPPAR	
)	CLPAR	
{	OPBRACE	
local	LOCAL	local
:	COLON	
int	INT	int
i	ID	i
;	SCOLON	
logic	LOGIC	logic
fim	ID	fim
;	SCOLON	
statements	STATEMENTS	statements
:	COLON	
write	WRITE	write
(	OPPAR	
" "	STRING	
,	COMMA	
"Palavra "	STRING	Palavra
,	COMMA	
" Num. de ocorr."	STRING	Num. de ocorr.
)	CLPAR	
;	SCOLON	

for	FOR	for
(	OPPAR	
i	ID	i
<-	ASSIGN	
1	INTCNT	1
;	SCOLON	
i	ID	i
<=	RELOP	LE
50	INTCNT	50
;	SCOLON	
i	ID	i
<-	ASSIGN	
i	ID	i
+	ADOP	MAIS
1	INTCNT	1
)	CLPAR	
write	WRITE	write
(	OPPAR	
"-"	STRING	-
)	CLPAR	
;	SCOLON	
for	FOR	for
(	OPPAR	
i	ID	i
<-	ASSIGN	

1	INTCNT	1
;	SCOLON	
i	ID	i
<=	RELOP	LE
ntab	ID	ntab
;	SCOLON	
i	ID	i
<-	ASSIGN	
i	ID	i
+	ADOP	MAIS
1	INTCNT	1
)	CLPAR	
{	OPBRACE	
write	WRITE	write
(	OPPAR	
"\n "	STRING	\n
)	CLPAR	
;	SCOLON	
fim	ID	fim
<-	ASSIGN	
false	FALSE	false
;	SCOLON	
for	FOR	for
(	OPPAR	
j	ID	j

<-	ASSIGN	
0	INTCNT	0
;	SCOLON	
!	NOT	
fim	ID	fim
;	SCOLON	
j	ID	j
<-	ASSIGN	
j	ID	j
+	ADOP	MAIS
1	INTCNT	1
)	CLPAR	
{	OPBRACE	
if	IF	if
(	OPPAR	
nomes	ID	nomes
[	OPBRAK	
i	ID	i
,	COMMA	
j	ID	j
]	CLBRAK	
=	RELOP	EQ
'\0'	CHARCT	\0
)	CLPAR	
fim	ID	fim

<-	ASSIGN	
true	TRUE	true
;	SCOLON	
else	ELSE	else
write	WRITE	write
(	OPPAR	
nomes	ID	nomes
[	OPBRAK	
i	ID	i
,	COMMA	
j	ID	j
]	CLBRAK	
)	CLPAR	
;	SCOLON	
}	CLBRACE	
write	WRITE	write
(	OPPAR	
" | "	STRING	|
,	COMMA	
nocorr	ID	nocorr
[	OPBRAK	
i	ID	i
]	CLBRAK	
)	CLPAR	
;	SCOLON	

}	CLBRACE	
}	CLBRACE	
main	MAIN	main
{	OPBRACE	
local	LOCAL	local
:	COLON	
int	INT	int
i	ID	i
,	COMMA	
posic	ID	posic
;	SCOLON	
char	CHAR	char
c	ID	c
;	SCOLON	
logic	LOGIC	logic
fim	ID	fim
;	SCOLON	
statements	STATEMENTS	statements
:	COLON	
ntab	ID	ntab
<-	ASSIGN	
0	INTCNT	0
;	SCOLON	
write	WRITE	write

(	OPPAR	
"Nova palavra? (s/n): "	STRING	Nova palavra? (s/n):
)	CLPAR	
;	SCOLON	
read	READ	read
(	OPPAR	
c	ID	c
)	CLPAR	
;	SCOLON	
while	WHILE	while
(	OPPAR	
c	ID	c
=	RELOP	EQ
's'	CHARCT	s
||	OR	
c	ID	c
=	RELOP	EQ
'S'	CHARCT	S
)	CLPAR	
{	OPBRACE	
write	WRITE	write
(	OPPAR	
"\nDigite a palavra: "	STRING	\nDigite a palavra:
)	CLPAR	
;	SCOLON	

fim	ID	fim
<-	ASSIGN	
false	FALSE	false
;	SCOLON	
for	FOR	for
(	OPPAR	
i	ID	i
<-	ASSIGN	
0	INTCNT	0
;	SCOLON	
!	NOT	
fim	ID	fim
;	SCOLON	
i	ID	i
<-	ASSIGN	
i	ID	i
+	ADOP	MAIS
1	INTCNT	1
)	CLPAR	
{	OPBRACE	
read	READ	read
(	OPPAR	
palavra	ID	palavra
[	OPBRAK	
i	ID	i

]	CLBRAK	
)	CLPAR	
;	SCOLON	
if	IF	if
(	OPPAR	
palavra	ID	palavra
[	OPBRAK	
i	ID	i
]	CLBRAK	
=	RELOP	EQ
'\n'	CHARCT	\n
)	CLPAR	
{	OPBRACE	
fim	ID	fim
<-	ASSIGN	
true	TRUE	true
;	SCOLON	
palavra	ID	palavra
[	OPBRAK	
i	ID	i
]	CLBRAK	
<-	ASSIGN	
'\0'	CHARCT	\0
;	SCOLON	
}	CLBRACE	

}	CLBRACE	
posic	ID	posic
<-	ASSIGN	
Procura	ID	Procura
(	OPPAR	
)	CLPAR	
;	SCOLON	
if	IF	if
(	OPPAR	
posic	ID	posic
>	RELOP	GT
0	INTCNT	0
)	CLPAR	
nocorr	ID	nocorr
[	OPBRAK	
posic	ID	posic
]	CLBRAK	
<-	ASSIGN	
nocorr	ID	nocorr
[	OPBRAK	
posic	ID	posic
]	CLBRAK	
+	ADOP	MAIS
1	INTCNT	1
;	SCOLON	

else	ELSE	else
call	CALL	call
Inserir	ID	Inserir
(	OPPAR	
~	NEG	
posic	ID	posic
,	COMMA	
i	ID	i
)	CLPAR	
;	SCOLON	
write	WRITE	write
(	OPPAR	
"\n\nNova palavra? (s/n): "	STRING	\n\nNova palavra? (s/n):
)	CLPAR	
;	SCOLON	
read	READ	read
(	OPPAR	
c	ID	c
)	CLPAR	
;	SCOLON	
}	CLBRACE	
call	CALL	call
ExibirTabela	ID	ExibirTabela
(	OPPAR	
)	CLPAR	

;	SCOLON	
}	CLBRACE	
}	CLBRACE	

APÊNDICE B – ANÁLISE LÉXICA DO CÓDIGO PARA EXERCÍCIO DE CASOS PROBLEMÁTICOS

Texto	**Tipo**	**Atributo**
"abc"	STRING	abc
"def"	STRING	def
sao	ID	sao
2	INTCT	2
strings	ID	strings
;	SCOLON	
2	INTCT	2
%	MULTOP	RESTO
3	INTCT	3
/	MULTOP	DIV
4	INTCT	4
+	ADOP	MAIS
5	INTCT	5
-	ADOP	MENOS
6	INTCT	6
*	MULTOP	VEZES
7	INTCT	7
sao	ID	sao
operacoes	ID	operacoes
;	SCOLON	
'	INVALIDO	'
\	INVALIDO	\

' '	CHARCT	
\	INVALIDO	\
'	INVALIDO	'
' '	CHARCT	
'	INVALIDO	'
sao	ID	sao
CHARCT	ID	CHARCT
;	SCOLON	
"ab\"c"	STRING	ab\"c
nao	ID	nao
eh	ID	eh
possivel	ID	possivel
escape	ID	escape
de	ID	de
aspas	ID	aspas
;	SCOLON	
nada	ID	nada
de	ID	de
á	INVALIDO	á
centos	ID	centos
;	SCOLON	
12.3	FLOATCT	12.3
eh	ID	eh
float	FLOAT	float
,	COMMA	

nao	ID	nao
int	INT	int
;	SCOLON	
<	RELOP	LT
eh	ID	eh
menor	ID	menor
,	COMMA	
<=	RELOP	LE
eh	ID	eh
menor	ID	menor
ou	ID	ou
igual	ID	igual
;	SCOLON	
>	RELOP	GT
eh	ID	eh
maior	ID	maior
,	COMMA	
>=	RELOP	GE
eh	ID	eh
maior	ID	maior
ou	ID	ou
igual	ID	igual
;	SCOLON	
em	ID	em
ID	ID	ID

,	COMMA	
soh	ID	soh
letras	ID	letras
_	INVALIDO	_
mais	ID	mais
_	INVALIDO	_
nada	ID	nada
;	SCOLON	
!=	RELOP	NE
eh	ID	eh
diferente	ID	diferente
,	COMMA	
!	NOT	
eh	ID	eh
negacao	ID	negacao
,	COMMA	
=	RELOP	EQ
eh	ID	eh
igualdade	ID	igualdade
;	SCOLON	
CHARCT	ID	CHARCT
'	INVALIDO	'
'	INVALIDO	'
nao	ID	nao
pode	ID	pode

ser	ID	ser
multi	ID	multi
-	ADOP	MENOS
linhas	ID	linhas
;	SCOLON	
strings	ID	strings
"	INVALIDO	"
"	INVALIDO	"
nao	ID	nao
podem	ID	podem
ser	ID	ser
multi	ID	multi
-	ADOP	MENOS
linhas	ID	linhas
tambem	ID	tambem
;	SCOLON	
do	DO	do
eh	ID	eh
keyword	ID	keyword
;	SCOLON	
doing	ID	doing
,	COMMA	
nao	ID	nao
;	SCOLON	
'	INVALIDO	'

\	INVALIDO	\
'	INVALIDO	'
;	SCOLON	
'	INVALIDO	'
'	INVALIDO	'
'	INVALIDO	'
;	SCOLON	
nao	ID	nao
sao	ID	sao
CHARCT	ID	CHARCT
,	COMMA	
mas	ID	mas
'\"	CHARCT	\'
eh	ID	eh