Gustavo P. Gouveia (6482819), Victor Lassance (6431325)

Relatório de Compiladores - Primeira Etapa - Construção de um analisador léxico

São Paulo

2013

Gustavo P. Gouveia (6482819), Victor Lassance (6431325)

Relatório de Compiladores - Primeira Etapa - Construção de um analisador léxico

Texto apresentado à Escola Politécnica da Universidade de São Paulo como requisito para a aprovação na disciplina Linguagens e Compiladores no quinto módulo acadêmico do curso de graduação em Engenharia de Computação, junto ao Departamento de Engenharia de Computação e Sistemas Digitais (PCS).

Universidade de São Paulo

Escola Politécnica

Engenharia de Computação - Curso Cooperativo

Professor: Ricardo Luis de Azevedo da Rocha

São Paulo

2013

Resumo

Este trabalho descreve a concepção e o desenvolvimento de um compilador utilizando a linguagem C. O escopo do compilador se limita a casos mais simples, porém simbólicos, e que servem ao aprendizado do processo de criação e teste de um compilador completo. A estrutura da linguagem escolhida para ser implementada se assemelha a própria estrutura do C, por facilidade de compreensão.

Palavras-chaves: Linguagens, Compiladores, Analisador Léxico.

Sumário

Sumário 3

1 Introdução 4

2 Questões 5

2.1 Questão 1 5
2.2 Questão 2 6
2.3 Questão 3 7
2.4 Questão 4 9
2.5 Questão 5 12
2.6 Questão 6 13
2.7 Questão 7 13
2.8 Questão 8 14
2.9 Questão 9 14
2.10 Questão 10 14

3 Conclusão 15

Referências 16

Apêndices 17

APÊNDICE A Transdutor do Analisador Léxico 18

APÊNDICE B Código em C da sub-rotina do Analisador Léxico 20

APÊNDICE C Código em C do método principal do Analisador Léxico 29

1 Introdução

TODO Introdução vlassance

2 Questões

A seguir, seguem as respostas às questões propostas pelo professor.

2.1 Questão 1

Quais são as funções do analisador léxico nos compiladores e interpretadores?

O analisador léxico atua como uma interface entre o reconhecedor sintático, que forma, normalmente, o núcleo do compilador, e o texto de entrada, convertendo a sequência de caracteres de que este se constitui em uma sequência de átomos.

Para a consecução de seus objetivos, o analisador léxico executa usualmente uma série de funções, todas de grande importância como infraestrutura para a operação das partes do compilador mais ligadas à tradução propriamente dita do texto-fonte. As principais funções são listadas abaixo:

- Extração e Classificação de Átomos;
  - Principal funcionalidade do analisador;
  - As classes de átomos mais usuais: identificadores, palavras reservadas, números inteiros sem sinal, números reais, strings, sinais de pontuação e de operação, caracteres especiais, símbolos compostos de dois ou mais caracteres especiais e comentários.
- Eliminação de Delimitadores e Comentários;
- Conversão numérica;
  - Conversão numérica de notações diversas em uma forma interna de representação para manipulação de pelos demais módulos do compilador.
- Tratamento de Identificadores;
  - Tratamento com auxílio de uma tabela de símbolos.
- Identificação de Palavras Reservadas;
  - Verificar se cada identificador reconhecido pertence a um conjunto de identificadores especiais.

- Recuperação de Erros;
- Listagens;
  - Geração de listagens do texto-fonte.
- Geração de Tabelas de Referências Cruzadas;
  - Geração de listagem indicativa dos símbolos encontrados, com menção à localização de todas as suas ocorrências no texto do programa-fonte.
- Definição e Expansão de Macros;
  - Pode ser realizado em um pré-processamento ou no analisador léxico. No caso do analisador, deve-se haver uma comunicação entres os analisadores léxico e sintático.
- Interação com o sistema de arquivos;
- Compilação Condicional;
- Controles de Listagens.
  - São os comandos que permitem ao programador que ligue e desligue opções de listagem, de coleta de símbolos em tabelas de referência cruzadas, de geração, e impressão de tais tabelas, de impressão de tabelas de símbolos do programa compilador, de tabulação e formatação das saídas impressas do programa-fonte.

2.2 Questão 2

Quais as vantagens e desvantagens da implementação do analisador léxico como uma fase separada do processamento da linguagem de programação em relação à sua implementação como sub-rotina que vai extraindo um átomo a cada chamada?

Geralmente, o gargalo encontrado durante a compilação de um programa sem otimização é a leitura de arquivos e a análise léxica. A separação, seja por pré-processamento ou por sub-rotina oferece o desacoplamento da leitura do arquivo e do resto do processo, facilitando a otimização da mesma.

A separação completa do analisador léxico possibilita a validação léxica de todos os tokens antes da Separando-se o analisador léxico do resto do compilador, é possivel otimizar esse módulo e obter um analisador léxico genérico que serviria a princípio para qualquer linguagem.

A desvantagem de se separar os dois é o desacoplamento da lógica e, por conseguinte, das informações disponíveis ao analisador sintático e semântico, informações estas que podem ser importantes no reconhecimento das classes dos tokens encontrados dependendo da linguagem a ser compilada.

Exemplo: Shell Script - O primeiro `echo` refere-se ao comando echo e o segundo refere-se ao primeiro argumento do comando.

```
1   echo echo
```

2.3 Questão 3

Defina formalmente, através de expressões regulares sobre o conjunto de caracteres ASCII, a sintaxe de cada um dos tipos de átomos a serem extraídos do texto-fonte pelo analisador léxico, bem como de cada um dos espaçadores e comentários.

- DELIM: `/[{}()\[\];]/`

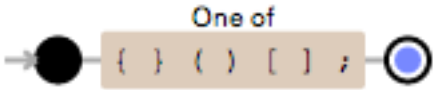


Figura 1 – Expressão Regular DELIM

- SPACE: `/[ \t\r\n\v\f]+/`

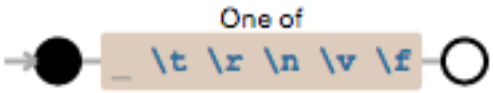


Figura 2 – Expressão Regular SPACE

- COMMENT: `/#[^\n]*/`

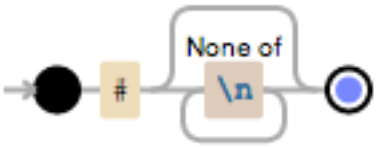


Figura 3 – Expressão Regular COMMENT

- IDENT: `/[a-zA-Z_][a-zA-Z0-9_]*/`

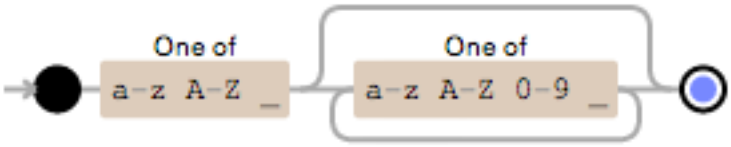


Figura 4 – Expressão Regular IDENT

- INTEGER: `/[0-9]+/`

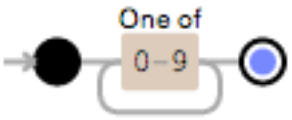


Figura 5 – Expressão Regular INTEGER

- FLOAT: `/[0-9]*\.[0-9]+/`

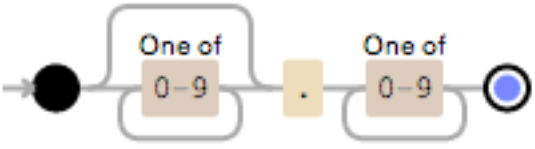


Figura 6 – Expressão Regular FLOAT

- CHAR: `/'(?:\\[0abtnvfre\\'"]|[\x20-\x5B\x5D-\x7E])'/`

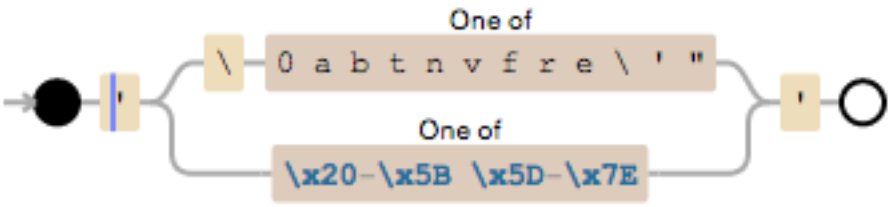


Figura 7 – Expressão Regular CHAR

- STRING: `/"(?:\\"|[^"])*"/`

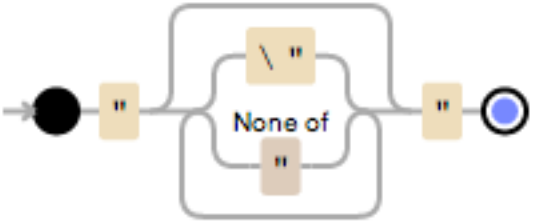


Figura 8 – Expressão Regular STRING

- OPER: `/[\+\-\*\/%=!<>][=]?/`

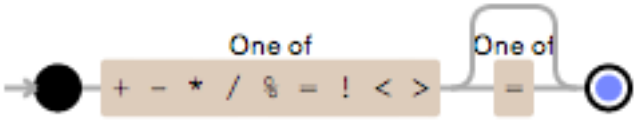


Figura 9 – Expressão Regular OPER

2.4 Questão 4

Converta cada uma das expressões regulares, assim obtidas, em autômatos finitos equivalentes que reconheçam as correspondentes linguagens por elas definidas.

- DELIM: `/[{}()\[\];]/`

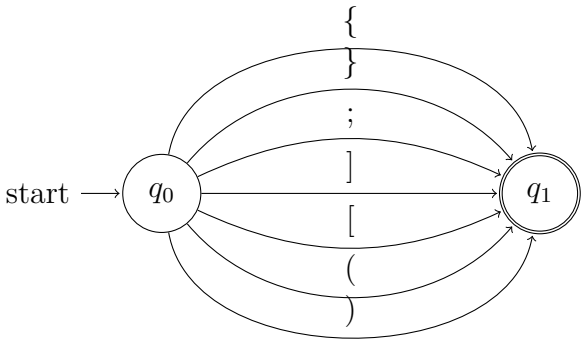


Figura 10 – Autômato finito DELIM

- SPACE: `/[ \t\r\n\v\f]+/`

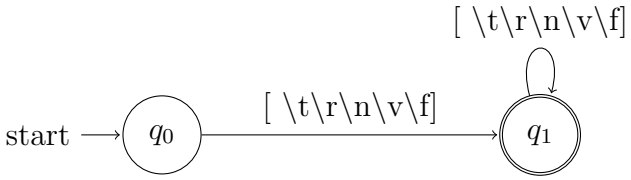


Figura 11 – Autômato finito SPACE

- COMMENT: `/#[^\n]*/`

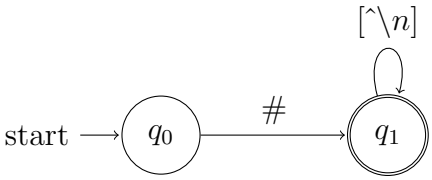


Figura 12 – Autômato finito COMMENT

- IDENT: `/[a-zA-Z_][a-zA-Z0-9_]*/`

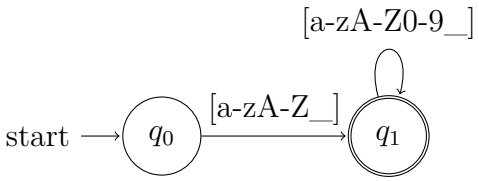


Figura 13 – Autômato finito IDENT

- INTEGER: `/[0-9]+/`

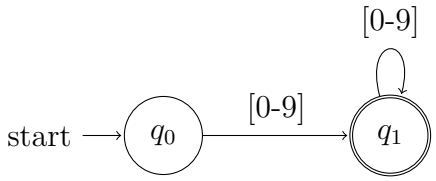


Figura 14 – Autômato finito INTEGER

- FLOAT: `/[0-9]*\.[0-9]+/`

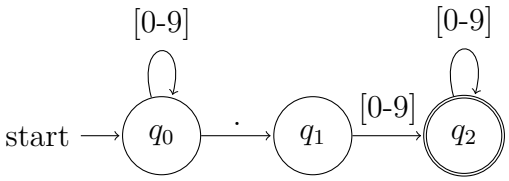


Figura 15 – Autômato finito FLOAT

- CHAR: `/'(?:\\[0abtnvfre\\'"]|[\x20-\x5B\x5D-\x7E])'/`

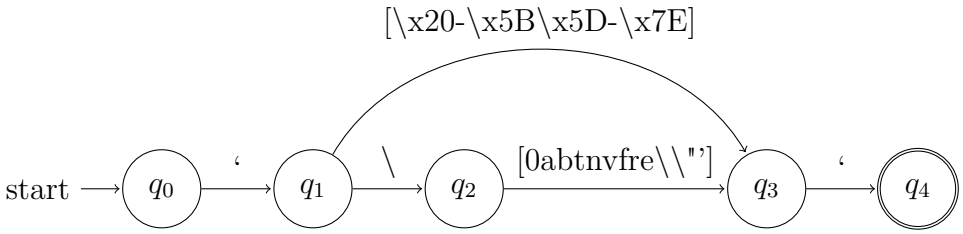


Figura 16 – Autômato finito CHAR

- STRING: `/"(?:\\"|[^"])*"/`

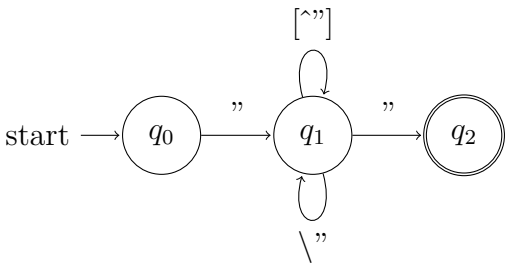


Figura 17 – Autômato finito STRING

- OPER: `/[\+\-\*\/%=!<>][=]?/`

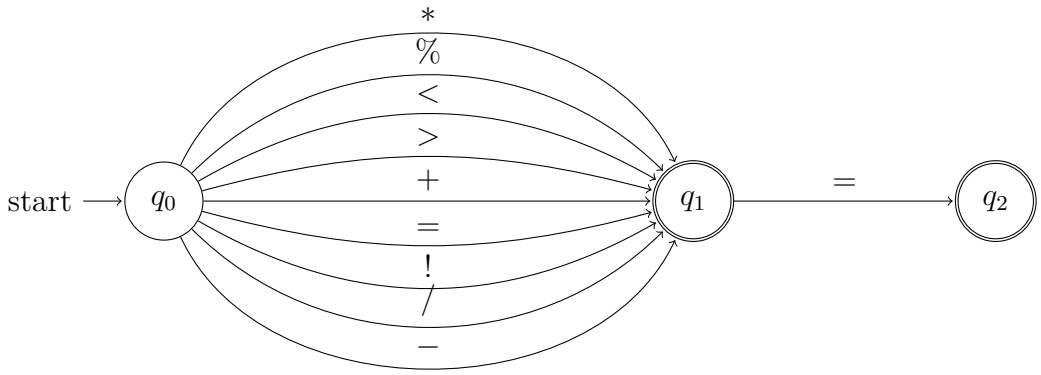
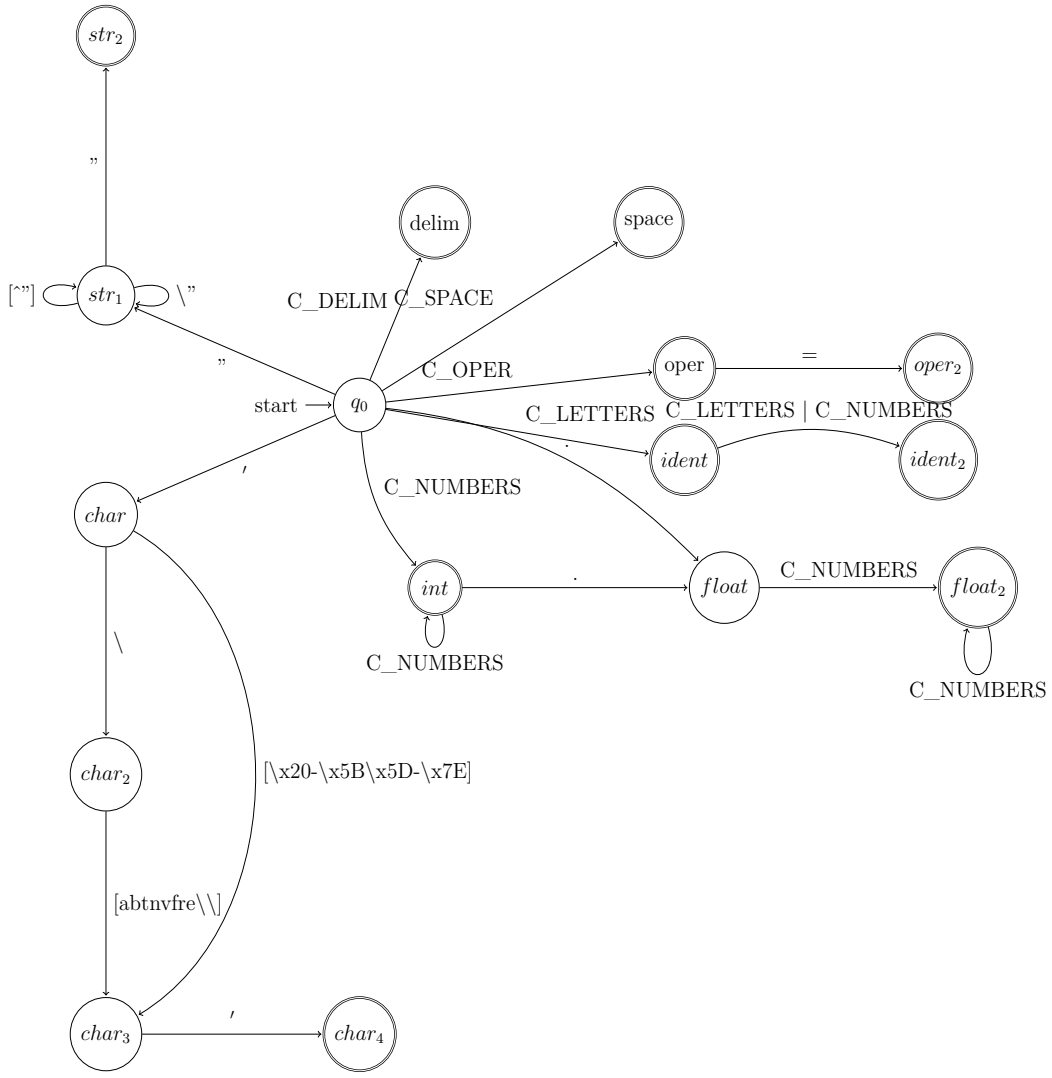


Figura 18 – Autômato finito OPER

2.5 Questão 5

Crie um autômato único que aceite todas essas linguagens a partir de um mesmo estado inicial, mas que apresente um estado final diferenciado para cada uma delas.

- C_DELIM = $[91, 93, 123, 125, 40, 41, 59]$
- C_SPACE = $[32, 9, 10, 11, 12, 13]$
- C_OPER = $[42, 37, 60, 62, 43, 61, 33, 47, 45]$
- C_LETTERS = $[65, \ldots, 90, 97, \ldots, 122, 95]$
- C_NUMBERS = $[48, 57]$



2.6 Questão 6

Transforme o autômato assim obtido em um transdutor, que emita como saída o átomo encontrado ao abandonar cada um dos estados finais para iniciar o reconhecimento de mais um átomo do texto.

O transdutor obtido a partir da transformação da questão 5 pode ser encontrado no apêndice A.

2.7 Questão 7

Converta o transdutor assim obtido em uma sub-rotina, escrita na linguagem de programação de sua preferência.

A sub-rotina escrita e testada pode ser encontrada no apêndice B. O código está comentado e seu funcionamento é explicado na questão 9.

2.8 Questão 8

Crie um programa principal que chame repetidamente a sub-rotina assim construída, e a aplique sobre um arquivo do tipo texto contendo o texto-fonte a ser analisado. Após cada chamada, esse programa principal deve imprimir as duas componentes do átomo extraído (o tipo e o valor do átomo encontrado).

O programa principal que utiliza as sub-rotinas pertencentes ao analisador léxico pode ser encontrada no apêndice C. O código está comentado e seu funcionamento é explicado na questão 9.

2.9 Questão 9

Relate detalhadamente o funcionamento do analisador léxico assim construído, incluindo no relatório: descrição teórica do programa; descrição da sua estrutura; descrição de seu funcionamento; descrição dos testes realizados e das saídas obtidas.

TODO gpg

2.10 Questão 10

Explique como enriquecer esse analisador léxico com um expansor de macros do tipo #DEFINE, não paramétrico nem recursivo, mas que permita a qualquer macro chamar outras macros, de forma não cíclica.

TODO

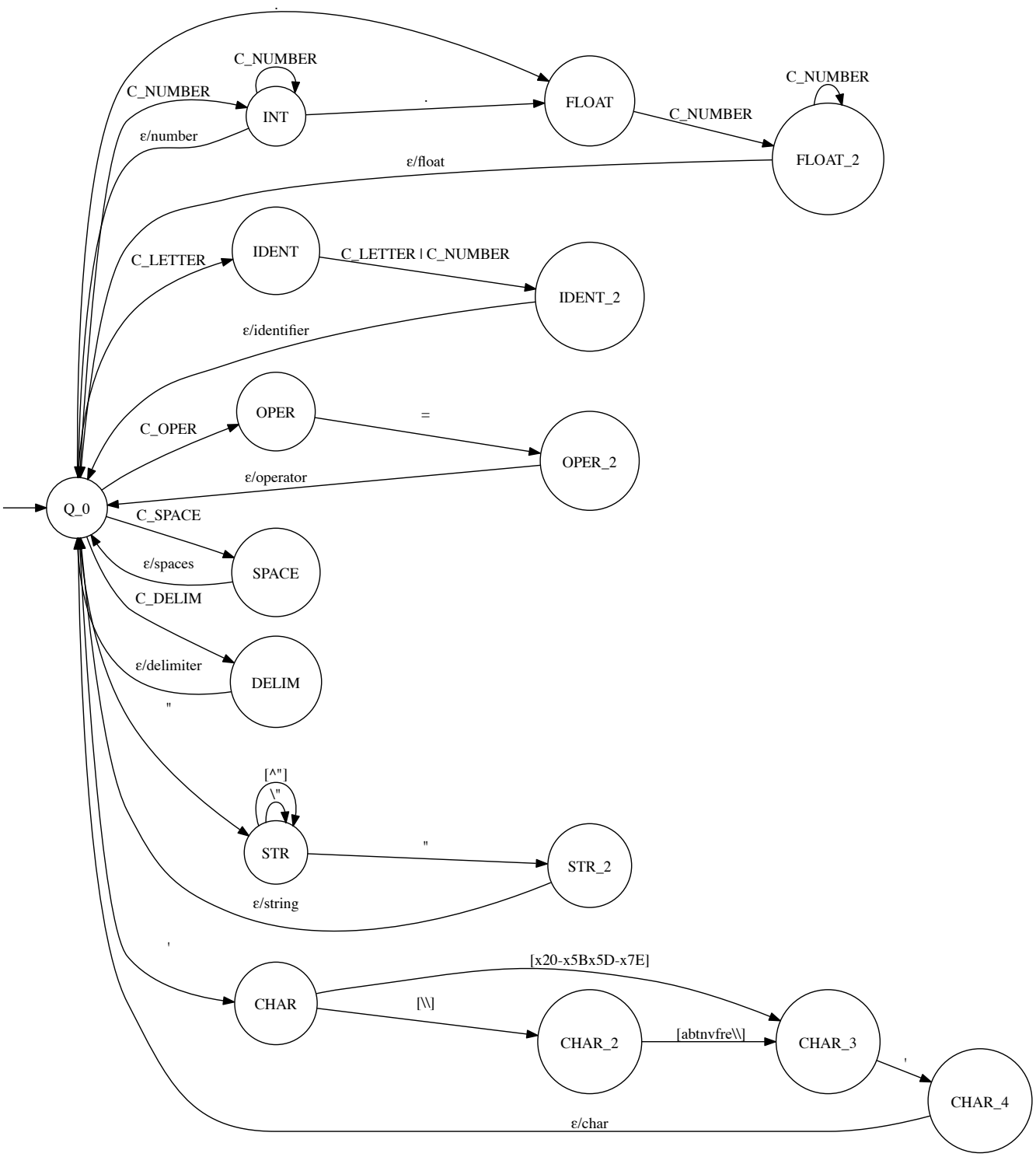
3 Conclusão

TODO Conclusão vlassance

Referências

Apêndices

APÊNDICE A – Transdutor do Analisador Léxico


.
C_NUMBER
C_NUMBER
INT
.
FLOAT
C_NUMBER
C_NUMBER
FLOAT_2
ε/number
ε/float
C_LETTER
IDENT
C_LETTER | C_NUMBER
IDENT_2
ε/identifier
C_OPER
OPER
=
OPER_2
ε/operator
Q_0
C_SPACE
ε/spaces
SPACE
C_DELIM
ε/delimiter
DELIM
"
[^"]
\"
STR
"
STR_2
ε/string
'
CHAR
[x20-x5Bx5D-x7E]
[\\]
CHAR_2
[abtnvfre\\]
CHAR_3
'
CHAR_4
ε/char

APÊNDICE B – Código em C da sub-rotina do Analisador Léxico

lex.h

```
1  #ifndef LEX_PCS2056
2
3  # define LEX_PCS2056
4
5  # define MAX_NUM_TRANSITIONS 50
6  # define MAX_NUM_STATES 50
7  # define MAXLENGTHSTATESTR 50
8  # define ENCODING_MAX_CHAR_NUM 256
9  # define MAX_SIZE_OF_A_TOKEN 2048
10 # define MAX_NUMBER_OF_KEYWORDS 256
11
12 typedef struct State {
13     char* name;
14     char* class_name;
15     int number_of_transitions;
16     long* masks[MAX_NUM_TRANSITIONS];
17     struct State* transitions[MAX_NUM_TRANSITIONS];
18 } State;
19
20 typedef struct Token {
21     long line;
22     long column;
23     long size;
24     char* class_name;
25     State* origin_state;
26     char* str;
27 } Token;
28
29 int _number_of_states;
30
```

```
31 State* state_table[MAX_NUM_STATES];
32 char buff_token[MAX_SIZE_OF_A_TOKEN];
33 long buff_token_end;
34
35
36 char* vkeywords[MAX_NUMBER_OF_KEYWORDS];
37 long vkeywords_size;
38
39 void initialize_lex();
40 int next_useful_token(FILE* f, Token** t);
41 void print_token(Token* t);
42
43 #endif
```

lex.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include "lex.h"
5
6  void state_from_name(char* statename, State** st) {
7      int i, size;
8      char* pch;
9      for (i = 0; i < _number_of_states; i++) {
10         if (strcmp(statename, state_table[i]->name) == 0) {
11             *st = state_table[i];
12             return;
13         }
14     }
15     state_table[_number_of_states] = malloc(sizeof(State));
16     state_table[_number_of_states]->name = malloc(
17         sizeof(char) * (strlen(statename) + 1)
18     );
19
20     strcpy(state_table[_number_of_states]->name, statename);
21
22     pch = strrchr(statename, '_');
23     if (!pch) {
24         size = strlen(statename);
```

```
25      } else {
26          size = pch - statename;
27      }
28      state_table[_number_of_states]->class_name = malloc(
29          sizeof(char) * (size + 1)
30      );
31      strncpy(state_table[_number_of_states]->class_name,
            statename, size);
32      *st = state_table[_number_of_states++];
33  }
34
35  void add_mask_to_state(State** from, State** to, long* mask) {
36      (*from)->masks[(*from)->number_of_transitions] = mask;
37      (*from)->transitions[(*from)->number_of_transitions++] = *to
            ;
38  }
39
40  void print_state(State* st) {
41      int i;
42      long maskterm, maskdepl, cod;
43      long masktermsize = sizeof(long) * 8;
44      printf("[%s]\n", st->name);
45      for (i = 0; i < st->number_of_transitions; i++) {
46          printf("␣");
47          for (cod = 0; cod < ENCODING_MAX_CHAR_NUM; cod++) {
48              maskterm = cod / masktermsize;
49              maskdepl = cod % masktermsize;
50              printf(
51                  "%c",
52                  (st->masks[i][maskterm] & (1L<<maskdepl))?'1':'0
                        ,
53              );
54          }
55          printf("\n␣");
56          for (cod = 0; cod < ENCODING_MAX_CHAR_NUM; cod++) {
57              maskterm = cod / masktermsize;
58              maskdepl = cod % masktermsize;
59              if (st->masks[i][maskterm] & (1L<<maskdepl)) {
60                  printf("%ld␣", cod);
```

```
61                  }
62              }
63              printf("␣->␣%s\n", st->transitions[i]->name);
64          }
65      }
66
67      void print_all_states() {
68          int i;
69          for (i = 0; i < _number_of_states; i++) {
70              print_state(state_table[i]);
71          }
72      }
73
74      int lex_parser_read_char(FILE* f) {
75          char fromname[MAXLENGTHSTATESTR];
76          char toname[MAXLENGTHSTATESTR];
77          long *mask;
78          char sep;
79          char c;
80          long cod;
81          long maskterm, maskdepl;
82          int i;
83          State *from;
84          State *to;
85
86          long masktermsize = sizeof(long) * 8; // number of byts on a
                long
87
88          if (fscanf(f, "␣%c", &sep) == EOF || sep == EOF) {
89              return 0;
90          }
91
92          mask = malloc(ENCODING_MAX_CHAR_NUM / (8));
93          for (i = 0; i < ENCODING_MAX_CHAR_NUM / (8 * sizeof(long));
               i++) {
94              mask[i] = (sep == '@')?(-1L):(0L);
95          }
96
97          while (fscanf(f, "%c", &c) && c != sep && c != EOF) {
```

```
98          cod = (long) c;
99          maskterm = cod / masktermsize;
100         maskdepl = cod % masktermsize;
101         mask[maskterm] |= (1L<<maskdepl);
102     }
103     fscanf(f, " %s", fromname);
104     state_from_name(fromname, &from);
105     fscanf(f, " %s", toname);
106     state_from_name(toname, &to);
107     add_mask_to_state(&from, &to, mask);
108     return 1;
109 }
110
111 void print_token(Token* t) {
112     printf("> [%s]", t->class_name);
113     printf(" >>%s<<", t->str);
114     printf(" at (%ld, %ld), with size %ld\n", t->line, t->column
            , t->size);
115 }
116
117 void find_next_state_from_char(char c, State** from, State** to)
        {
118     long masktermsize = sizeof(long) * 8; // number of byts on a
            long
119     long cod, maskterm, maskdepl;
120     int i;
121     (*to) = NULL;
122     cod = (long) c;
123     maskterm = cod / masktermsize;
124     maskdepl = cod % masktermsize;
125     for (i = 0; i < (*from)->number_of_transitions; i++) {
126         if ((*from)->masks[i][maskterm] & (1L<<maskdepl)) {
127             (*to) = (*from)->transitions[i];
128             break;
129         }
130     }
131 }
132
133 int next_useful_token(FILE* f, Token** t) {
```

```
134     int res, i;
135
136     do {
137         res = next_token(f, t);
138     } while(
139         *t != NULL &&
140         res &&
141         strcmp((*t)->origin_state->class_name, "SPACE") == 0
142     );
143
144     if (!res || *t == NULL){
145         return res;
146     }
147
148     if (strcmp((*t)->origin_state->class_name, "IDENT") == 0) {
149         for (i = 0; i < vkeywords_size; i++) {
150             if (strcmp((*t)->str, vkeywords[i]) == 0) {
151                 break;
152             }
153         }
154         if (i == vkeywords_size) {
155             (*t)->class_name = malloc(6 * sizeof(char));
156             strcpy((*t)->class_name, "IDENT");
157         } else {
158             (*t)->class_name = malloc(9 * sizeof(char));
159             strcpy((*t)->class_name, "RESERVED");
160         }
161     } else {
162         (*t)->class_name = malloc(
163             (strlen((*t)->origin_state->class_name) + 1) *
                sizeof(char)
164         );
165         strcpy((*t)->class_name, (*t)->origin_state->class_name)
            ;
166     }
167     // to be sure that this will not be used
168     (*t)->origin_state = NULL;
169     return res;
170 }
```

```
171
172 int next_token(FILE* f, Token** t) {
173     static State *current_state = NULL;
174     static long cline = 1;
175     static long ccolumn = 0;
176     static long line = 1;
177     static long column = 1;
178     static char tmpend = 1;
179     char next_c;
180
181     State* next_state;
182
183     if (tmpend == EOF) {
184         (*t) = NULL;
185         return 1;
186     }
187     if (current_state == NULL) {
188         state_from_name("Q0", &current_state);
189         buff_token_end = 0;
190         buff_token[0] = '\0';
191     }
192     do {
193         tmpend = fscanf(f, "%c", &next_c);
194         if (next_c == '\n') {
195             cline++;
196             ccolumn = 0;
197         } else {
198             if (ccolumn < 0) {
199                 ccolumn = 1;
200             } else {
201                 ccolumn++;
202             }
203         }
204         next_state = NULL;
205         find_next_state_from_char(next_c, &current_state, &
                next_state);
206         if (next_state != NULL && strcmp(next_state->name, "Q0")
                == 0){
207             (*t) = malloc(sizeof(Token));
```

```
208             (*t)->str = malloc(sizeof(char) * (strlen(buff_token
                    ) + 1L));
209             strcpy((*t)->str, buff_token);
210             (*t)->line = line;
211             (*t)->column = column;
212             (*t)->origin_state = current_state;
213             (*t)->size = strlen(buff_token);
214             find_next_state_from_char(next_c, &next_state, &
                    current_state);
215             column = ccolumn;
216             line = cline;
217             buff_token[0] = next_c;
218             buff_token[1] = '\0';
219             buff_token_end = 1;
220             if (current_state == NULL) {
221                 fprintf(
222                     stderr,
223                     "buff_token: <%s>, error at line %ld column %
                            ld\n",
224                     buff_token,
225                     cline,
226                     ccolumn
227                 );
228                 return 0;
229             }
230             return 1;
231         } else {
232             buff_token[buff_token_end++] = next_c;
233             buff_token[buff_token_end] = '\0';
234         }
235 
236         if (next_state == NULL) {
237             fprintf(
238                 stderr,
239                 "buff_token: <%s>, error at line %ld column %ld\
                        n",
240                 buff_token,
241                 cline,
242                 ccolumn
```

```
243                 );
244                 return 1;
245             }
246             current_state = next_state;
247         } while (tmpend != EOF);
248         (*t) = NULL;
249         return 0;
250 }
251
252 void initialize_lex() {
253         FILE *lex_file, *keywords_file;
254         vkeywords_size = 0;
255
256         lex_file = fopen("./languagefiles/lang.lex", "r");
257         keywords_file = fopen("./languagefiles/keywords.txt", "r");
258
259         while (lex_parser_read_char(lex_file)) {
260         }
261         while (fscanf(keywords_file, "␣%s", buff_token) != EOF) {
262             vkeywords[vkeywords_size] = malloc(sizeof(char) * (
                    strlen(buff_token) + 1L));
263             strcpy(vkeywords[vkeywords_size++], buff_token);
264         }
265         //print_all_states();
266 }
```

APÊNDICE C – Código em C do método principal do Analisador Léxico

```c
#include <stdio.h>
#include "lex.h"

int main(int argc, char *argv[]) {
    FILE *input_file;
    Token* tk;
    _number_of_states = 0;
        char abc = '';

    if (argc <= 1) {
        fprintf(stderr, "Usage:\n");
        fprintf(stderr, "  %s <input file>\n", argv[0]);
        return 1;
    }

    initialize_lex();

    input_file = fopen(argv[1], "r");

    while (next_useful_token(input_file, &tk) && tk != NULL) {
       print_token(tk);
    }
    if (tk == NULL)
        return 0;
    return 1;
}
```