Gustavo Pacianotto Gouveia (NUSP 6482819), Victor Lassance (NUSP 6431325)

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

São Paulo

2013

Gustavo Pacianotto Gouveia (NUSP 6482819), Victor Lassance (NUSP 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

1 Introdução

TODO Introdução

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. 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 analizador 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 échoŕefere-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: `/[{}()\[\];]/`

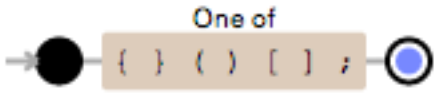


Figura 1 – Expressão Regular DELIM

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

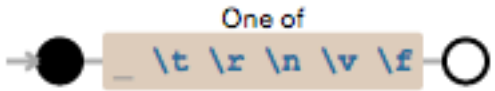


Figura 2 – Expressão Regular SPACE

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

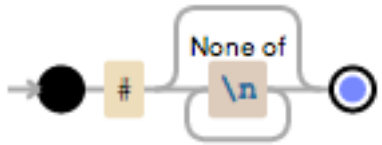


Figura 3 – Expressão Regular COMMENT

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

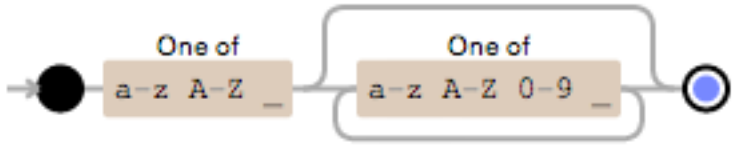


Figura 4 – Expressão Regular IDENT

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

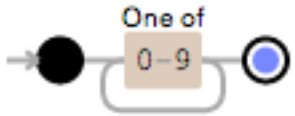


Figura 5 – Expressão Regular INTEGER

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

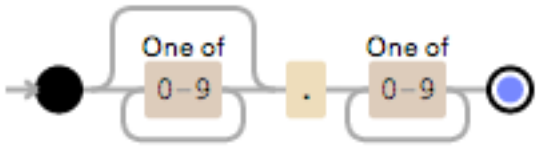


Figura 6 – Expressão Regular FLOAT

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

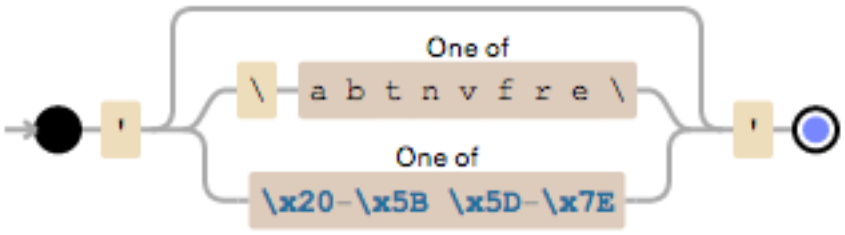


Figura 7 – Expressão Regular CHAR

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

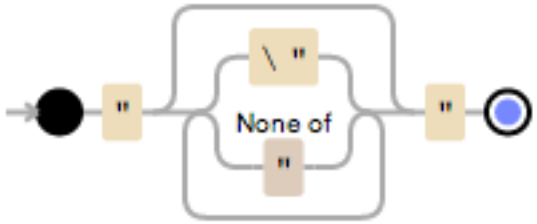


Figura 8 – Expressão Regular STRING

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

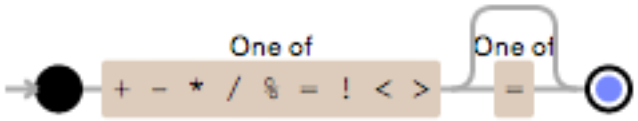


Figura 9 – Expressão Regular OPER

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

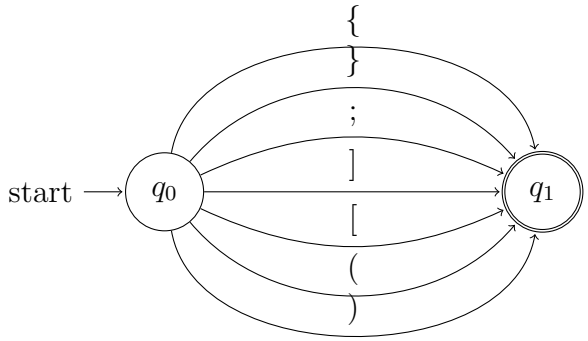


Figura 10 – Autômato finito DELIM

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

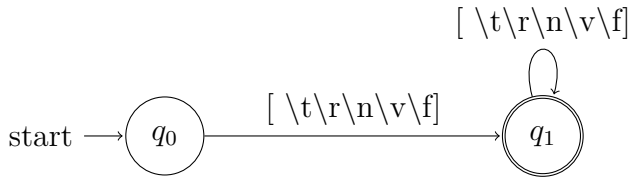


Figura 11 – Autômato finito SPACE

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

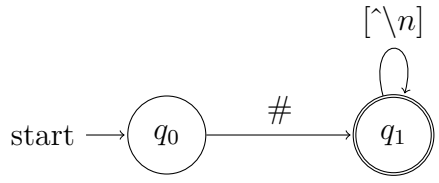


Figura 12 – Autômato finito COMMENT

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

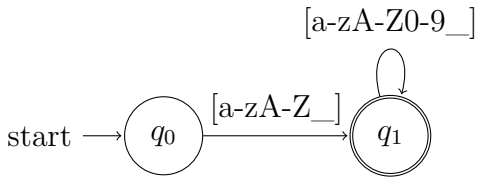


Figura 13 – Autômato finito IDENT

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

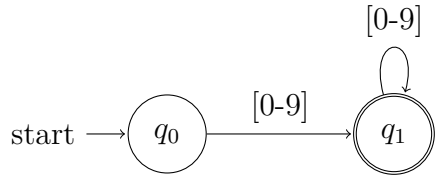


Figura 14 – Autômato finito INTEGER

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

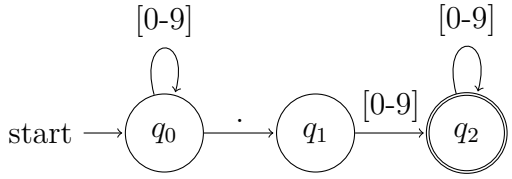


Figura 15 – Autômato finito FLOAT

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

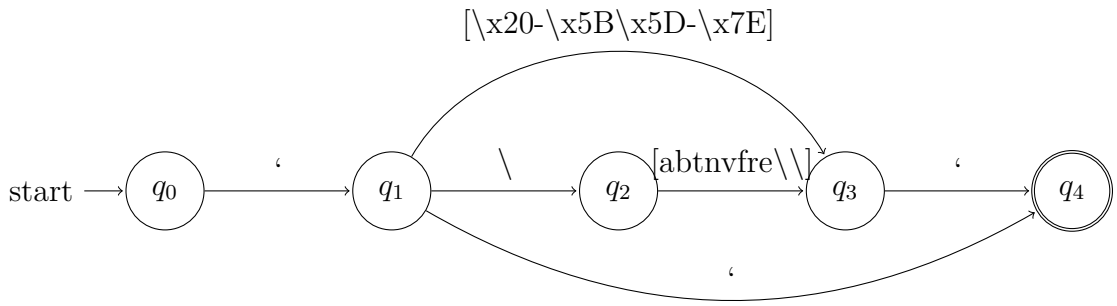


Figura 16 – Autômato finito CHAR

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

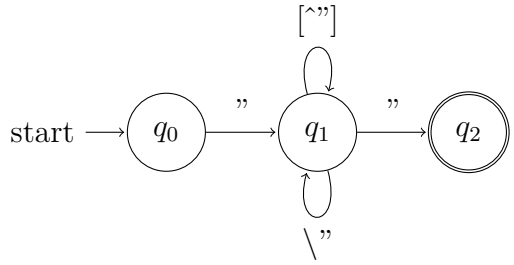


Figura 17 – Autômato finito STRING

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

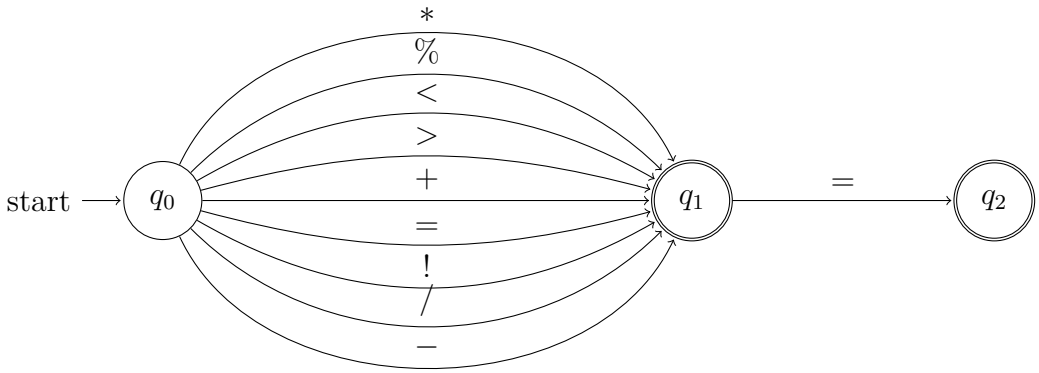


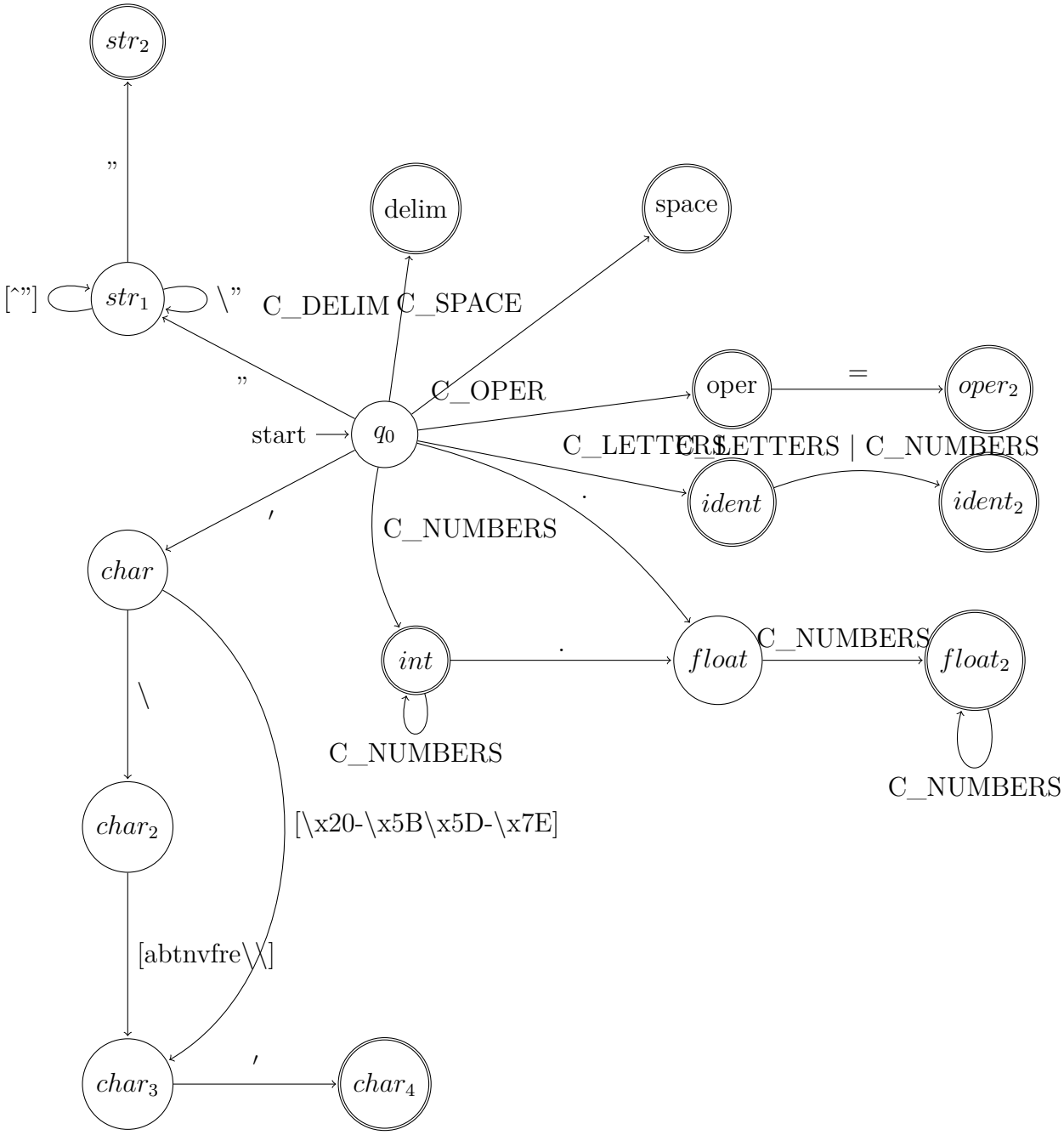
Figura 18 – Autômato finito OPER

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 = $[48, 57]$


str_2
"
delim
space
[^"]
str_1
\"
C_DELIM
C_SPACE
"
C_OPER
oper
=
$oper_2$
start
q_0
C_LETTERS
C_LETTERS | C_NUMBERS
.
$ident$
$ident_2$
'
C_NUMBERS
$char$
int
.
$float$
C_NUMBERS
$float_2$
\
C_NUMBERS
C_NUMBERS
$char_2$
[\x20-\x5B\x5D-\x7E]
[abtnvfre\\]
$char_3$
'
$char_4$

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 pode ser encontrado como apêndice A.

2.7 Questão 7

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

TODO

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).

TODO

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

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

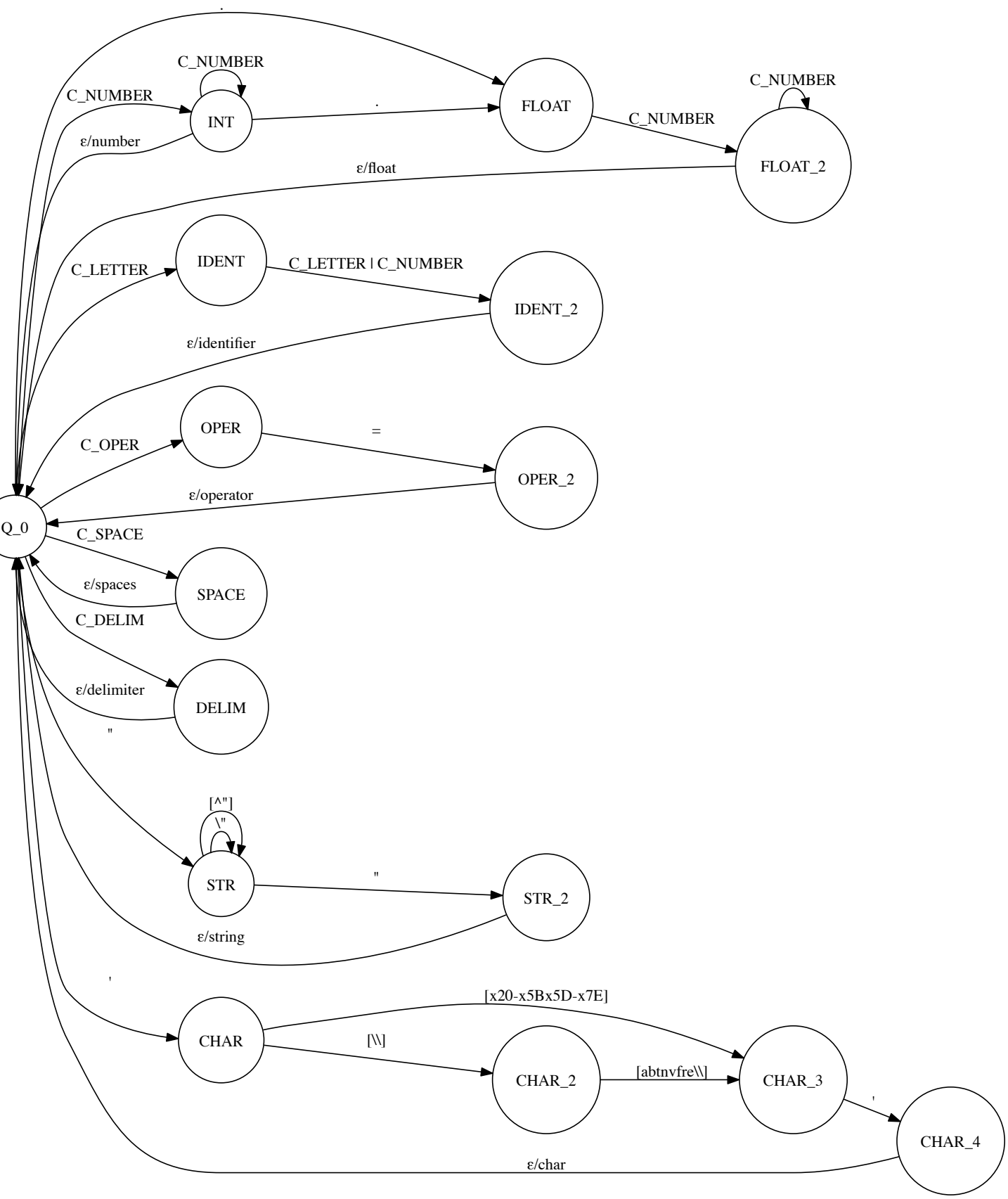
3 Conclusão

TODO Conclusão

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
SPACE
ε/spaces
C_DELIM
DELIM
ε/delimiter
"
[^"]
\"
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

```
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     strcpy(state_table[_number_of_states]->name, statename);
20     state_table[_number_of_states]->class_name = malloc(
```

```
21          sizeof(char) * (strlen(statename) + 1)
22      );
23      pch = strrchr(statename, '_');
24      if (!pch) {
25          size = strlen(statename);
26      } else {
27          size = pch - statename;
28      }
29      strncpy(state_table[_number_of_states]->class_name, statename, size);
30      *st = state_table[_number_of_states++];
31  }
32
33  void add_mask_to_state(State** from, State** to, long* mask) {
34      (*from)->masks[(*from)->number_of_transitions] = mask;
35      (*from)->transitions[(*from)->number_of_transitions++] = *to;
36  }
37
38  void print_state(State* st) {
39      int i;
40      long maskterm, maskdepl, cod;
41      long masktermsize = sizeof(long) * 8;
42      printf("[%s]\n", st->name);
43      for (i = 0; i < st->number_of_transitions; i++) {
44          printf("␣");
45          for (cod = 0; cod < ENCODING_MAX_CHAR_NUM; cod++) {
46              maskterm = cod / masktermsize;
47              maskdepl = cod % masktermsize;
48              printf(
```

```
49                      "%c",
50                      (st->masks[i][maskterm] & (1L<<maskdepl))?'1':'0'
51                  );
52              }
53              printf("\n ");
54              for (cod = 0; cod < ENCODING_MAX_CHAR_NUM; cod++) {
55                  maskterm = cod / masktermsize;
56                  maskdepl = cod % masktermsize;
57                  if (st->masks[i][maskterm] & (1L<<maskdepl)) {
58                      printf("%ld ", cod);
59                  }
60              }
61              printf(" -> %s\n", st->transitions[i]->name);
62          }
63      }
64
65      void print_all_states() {
66          int i;
67          for (i = 0; i < _number_of_states; i++) {
68              print_state(state_table[i]);
69          }
70      }
71
72      int lex_parser_read_char(FILE* f) {
73          char fromname[MAXLENGTHSTATESTR];
74          char toname[MAXLENGTHSTATESTR];
75          long *mask;
76          char sep;
```

```
77      char c;
78      long cod;
79      long maskterm, maskdepl;
80      int i;
81      State *from;
82      State *to;
83
84      long masktermsize = sizeof(long) * 8; // number of byts on a long
85
86      if (fscanf(f, " %c", &sep) == EOF || sep == EOF) {
87          return 0;
88      }
89
90      mask = malloc(ENCODING_MAX_CHAR_NUM / (8));
91      for (i = 0; i < ENCODING_MAX_CHAR_NUM / (8 * sizeof(long)); i++) {
92          mask[i] = (sep == '@')?(-1L):(0L);
93      }
94
95      while (fscanf(f, "%c", &c) && c != sep && c != EOF) {
96          cod = (long) c;
97          maskterm = cod / masktermsize;
98          maskdepl = cod % masktermsize;
99          mask[maskterm] |= (1L<<maskdepl);
100     }
101     fscanf(f, " %s", fromname);
102     state_from_name(fromname, &from);
103     fscanf(f, " %s", toname);
104     state_from_name(toname, &to);
```

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

```
133
134     do {
135         res = next_token(f, t);
136     } while(
137         *t != NULL &&
138         res &&
139         strcmp((*t)->origin_state->class_name, "SPACE") == 0
140     );
141
142     if (!res || *t == NULL){
143         return res;
144     }
145
146     if (strcmp((*t)->origin_state->class_name, "IDENT") == 0) {
147         for (i = 0; i < vkeywords_size; i++) {
148             if (strcmp((*t)->str, vkeywords[i]) == 0) {
149                 break;
150             }
151         }
152         if (i == vkeywords_size) {
153             (*t)->class_name = malloc(6 * sizeof(char));
154             strcpy((*t)->class_name, "IDENT");
155         } else {
156             (*t)->class_name = malloc(9 * sizeof(char));
157             strcpy((*t)->class_name, "RESERVED");
158         }
159     } else {
160         (*t)->class_name = malloc(
```

```
161             (strlen((*t)->origin_state->class_name) + 1) * sizeof(char)
162         );
163         strcpy((*t)->class_name, (*t)->origin_state->class_name);
164     }
165     // to be sure that this will not be used
166     (*t)->origin_state = NULL;
167     return res;
168 }
169
170 int next_token(FILE* f, Token** t) {
171     static State *current_state = NULL;
172     static long cline = 1;
173     static long ccolumn = 0;
174     static long line = 1;
175     static long column = 1;
176     static char tmpend = 1;
177     char next_c;
178
179     State* next_state;
180
181     if (tmpend == EOF) {
182         (*t) = NULL;
183         return 1;
184     }
185     if (current_state == NULL) {
186         state_from_name("Q0", &current_state);
187         buff_token_end = 0;
188         buff_token[0] = '\0';
```

```
189     }
190     do {
191         tmpend = fscanf(f, "%c", &next_c);
192         if (next_c == '\n') {
193             cline++;
194             ccolumn = 0;
195         } else {
196             if (ccolumn < 0) {
197                 ccolumn = 1;
198             } else {
199                 ccolumn++;
200             }
201         }
202         next_state = NULL;
203         find_next_state_from_char(next_c, &current_state, &next_state);
204         if (next_state != NULL && strcmp(next_state->name, "Q0") == 0){
205             (*t) = malloc(sizeof(Token));
206             (*t)->str = malloc(sizeof(char) * (strlen(buff_token) + 1L));
207             strcpy((*t)->str, buff_token);
208             (*t)->line = line;
209             (*t)->column = column;
210             (*t)->origin_state = current_state;
211             (*t)->size = strlen(buff_token);
212             find_next_state_from_char(next_c, &next_state, &current_state);
213             column = ccolumn;
214             line = cline;
215             buff_token[0] = next_c;
216             buff_token[1] = '\0';
```

```
217             buff_token_end = 1;
218             if (current_state == NULL && next_c != '\0') {
219                 printf(
220                     "buff_token: <%s>, error at line %ld column %ld\n",
221                     buff_token,
222                     cline,
223                     ccolumn
224                 );
225                 return 0;
226             }
227             return 1;
228         } else {
229             buff_token[buff_token_end++] = next_c;
230             buff_token[buff_token_end] = '\0';
231         }
232 
233         if (next_state == NULL) {
234            printf(
235                 "buff_token: <%s>, error at line %ld column %ld\n",
236                 buff_token,
237                 cline,
238                 ccolumn
239             );
240             return 0;
241         }
242         current_state = next_state;
243     } while (tmpend != EOF);
244     (*t) = NULL;
```

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

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

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

int main() {
    FILE *input_file;
    Token* tk;
    _number_of_states = 0;
    initialize_lex();

    input_file = fopen("../languagefiles/ex.czar", "r");

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