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

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

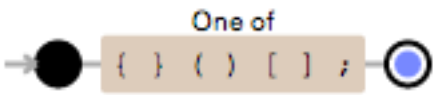


Figura 1 – Expressão Regular DELIM

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

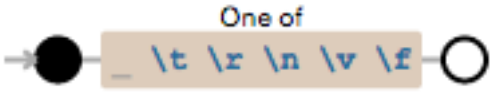


Figura 2 – Expressão Regular SPACE

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

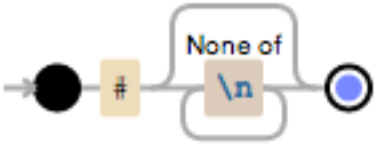


Figura 3 – Expressão Regular COMMENT

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

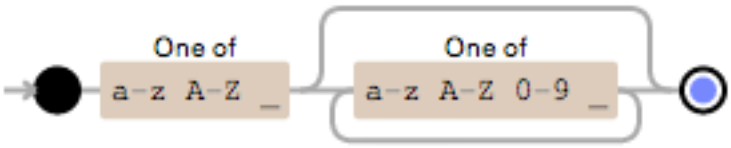


Figura 4 – Expressão Regular IDENT

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

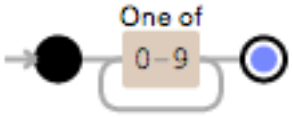


Figura 5 – Expressão Regular INTEGER

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

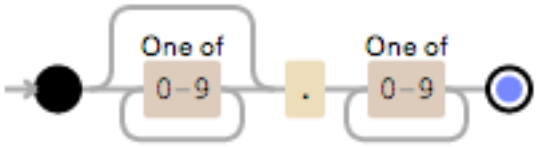


Figura 6 – Expressão Regular FLOAT

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

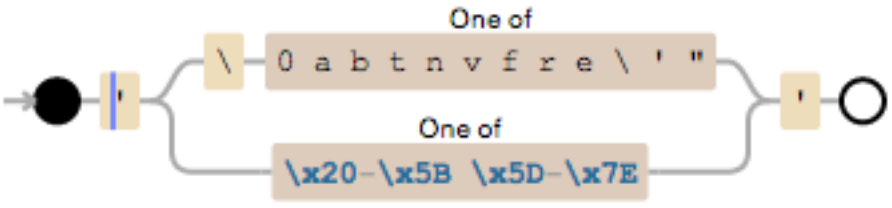


Figura 7 – Expressão Regular CHAR

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

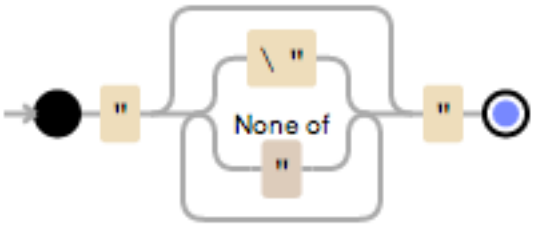


Figura 8 – Expressão Regular STRING

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

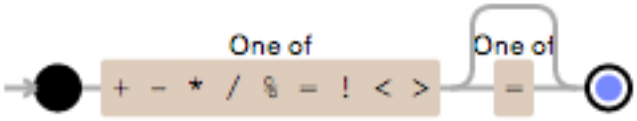


Figura 9 – Expressão Regular OPER

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

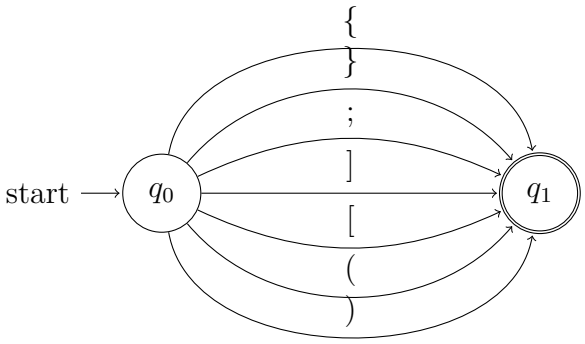


Figura 10 – Autômato finito DELIM

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

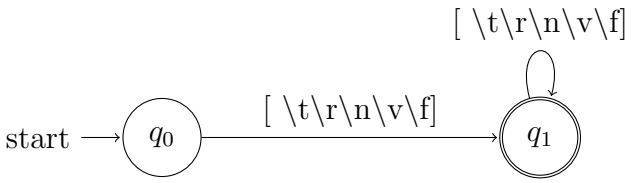


Figura 11 – Autômato finito SPACE

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

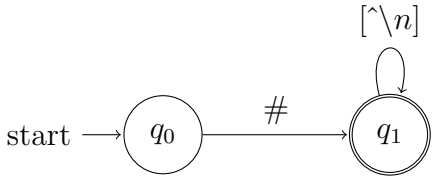


Figura 12 – Autômato finito COMMENT

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

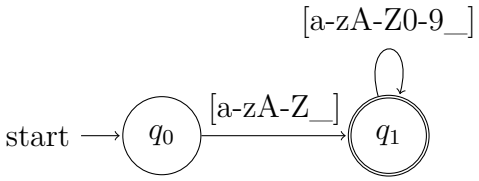


Figura 13 – Autômato finito IDENT

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

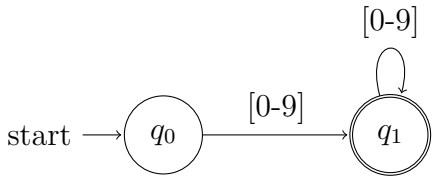


Figura 14 – Autômato finito INTEGER

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

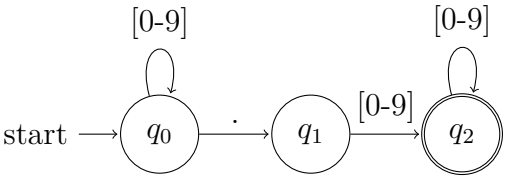


Figura 15 – Autômato finito FLOAT

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

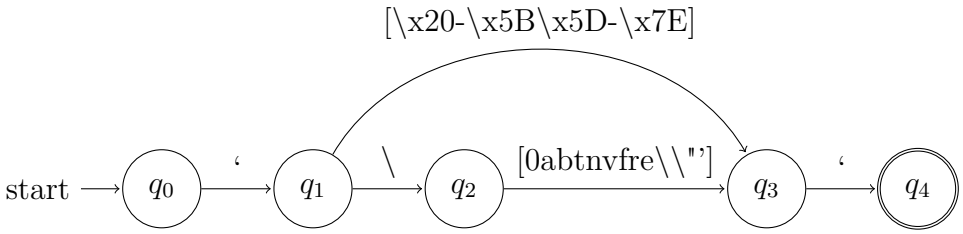


Figura 16 – Autômato finito CHAR

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

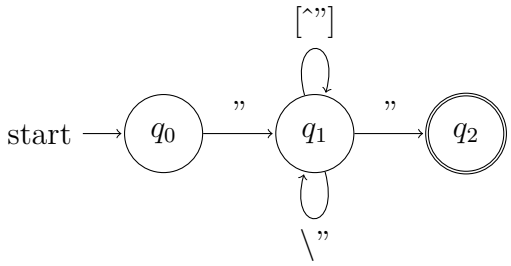


Figura 17 – Autômato finito STRING

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

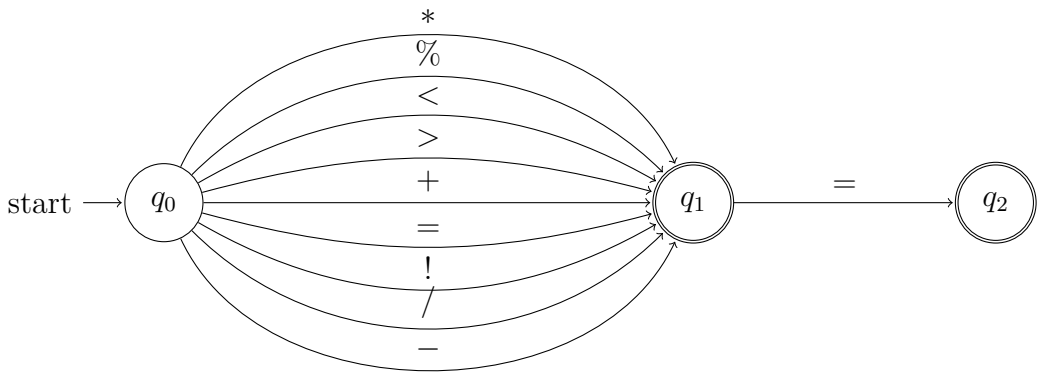
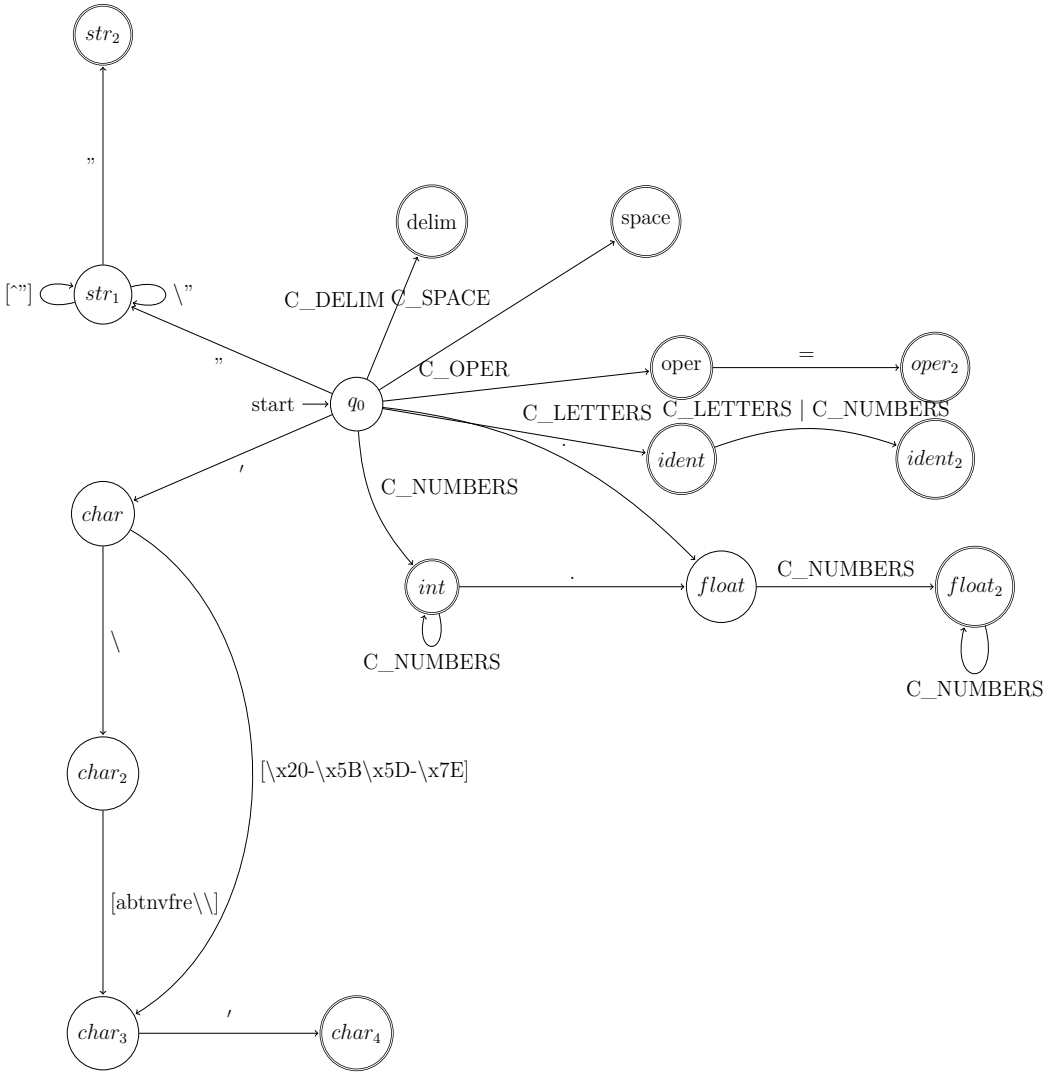


Figura 18 – Autômato finito OPER

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.

O analisador léxico lê um arquivo de configuração da máquina de estados (transdutor) o mesmo pode ser comparado à seguinte regex:

```
(.)([^\1]*)\1\s*([A-Za-z]+(:?_[0-9]+)?)\s*([A-Za-z]+(:?_[0-9]+)?)
```

Cada linha possui uma lista de caracteres delimitados por um caractere especial (por exemplo '+' ou '`#`') e dois identificadores que designam os estados inicial e final da transição. O caractere `@` designa todas as transições, este é usado principalmente para encaminhar qualquer aceitação final de um sub-autômato ao estado `Q0`, para então ser tratado normalmente.

Após a leitura do arquivo de configuração (e de um arquivo com *keywords*). Faz-se a leitura do arquivo fonte, por meio do transdutor, percorre-se o arquivo fonte token a token. Uma função `next_useful_token` oferece o não retorno dos tokens de espaço, tal qual a quebra de linha e espaços normais. O motor do lex também substitui classes `IDENT` em `RESERVED` se a palavra se encontra na lista de identificadores reservados.

Uma `hashtable` está sendo desenvolvida e será integrada nas próximas versões do compilador.

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.

Uma forma de permitir a utilização de macros seria o de realizar o pré-processamento, substituindo no código todas as macros pelos seus valores. Caso isso não seja desejado, também é possível acrescentar o tratamento de macros no analisador léxico, como explicitado na questão.

A maneira mais prática seria de se armazenar todas as macros em um vetor ou *hash table*, de forma similar a como é feito com as palavras reservadas. Ao se encontrar um identificador de macro, deve-se adicionar a um buffer o conteúdo da macro e deve-se processar o buffer até que o mesmo termine, antes de retornar a leitura do arquivo. Caso haja um identificador de macro dentro da definição de uma macro, pode-se substituir o identificador pela sua definição dentro do buffer que já está sendo lido, pra facilitar a lógica.

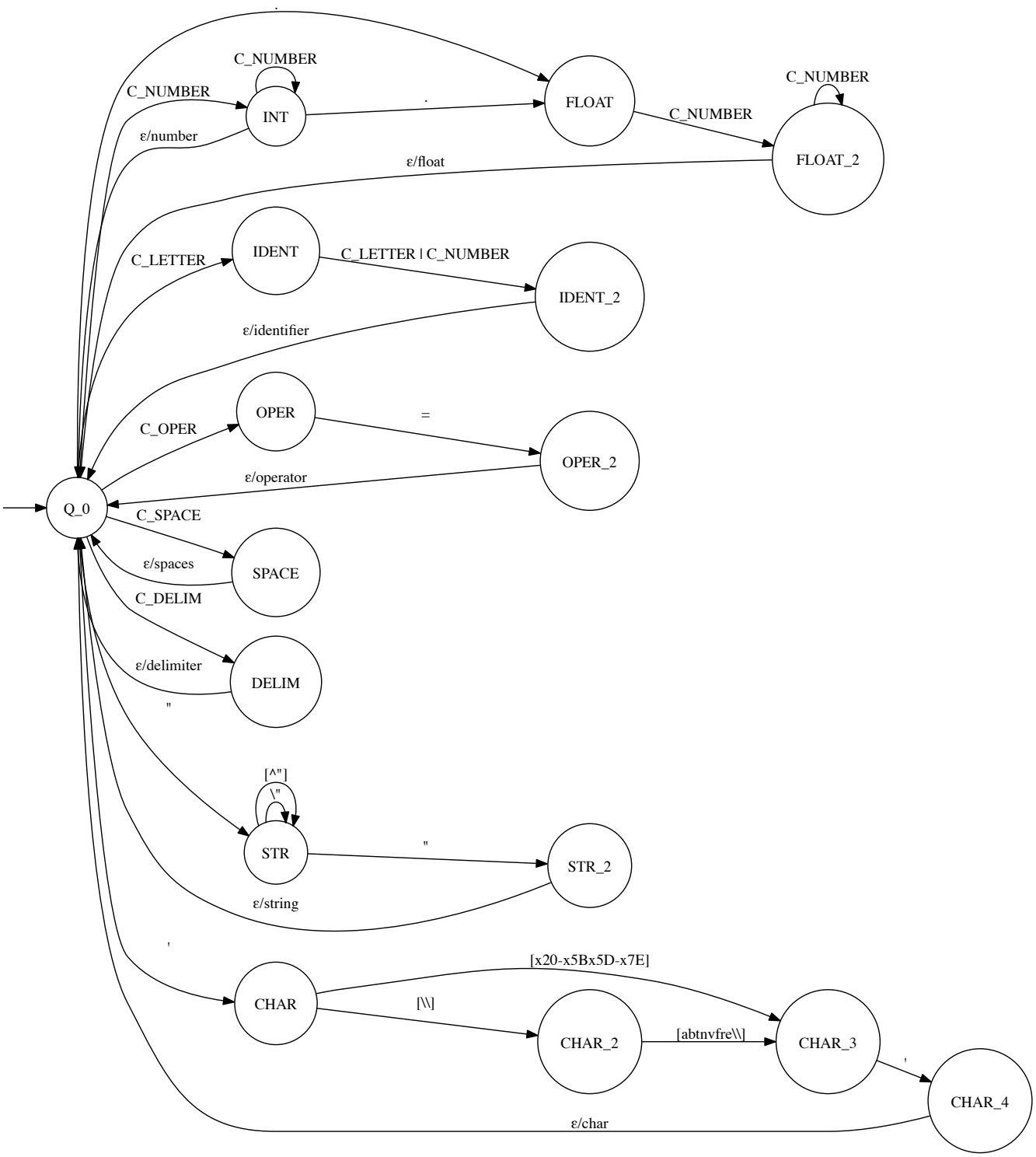
Uma possível solução para não tratar macros como casos únicos de utilização de buffers de leitura, pode-se ler um conjunto de caracteres a cada vez do arquivo e sempre armazenar em um buffer, lendo o arquivo novamente somente quando o buffer estiver vazio. Dessa forma, a macro nada mais será que uma substituição de um identificador por uma definição.

3 Conclusão

TODO Conclusão vlassance

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  /**
7   * Printing procedures
8   */
9  void print_state(State* st) {
10     int i;
11     long maskterm, maskdepl, cod;
12     long masktermsize = sizeof(long) * 8;
13     printf("[%s]\n", st->name);
14     for (i = 0; i < st->number_of_transitions; i++) {
15         printf("␣");
16         for (cod = 0; cod < ENCODING_MAX_CHAR_NUM; cod++) {
17             maskterm = cod / masktermsize;
18             maskdepl = cod % masktermsize;
19             printf(
20                 "%c",
21                 (st->masks[i][maskterm] & (1L<<maskdepl))?'1':'0
                       ,
22             );
23         }
24         printf("\n␣");
25         for (cod = 0; cod < ENCODING_MAX_CHAR_NUM; cod++) {
```

```
26              maskterm = cod / masktermsize;
27              maskdepl = cod % masktermsize;
28              if (st->masks[i][maskterm] & (1L<<maskdepl)) {
29                  printf("%ld ", cod);
30              }
31          }
32          printf(" -> %s\n", st->transitions[i]->name);
33      }
34  }
35
36  void print_all_states() {
37      int i;
38      for (i = 0; i < _number_of_states; i++) {
39          print_state(state_table[i]);
40      }
41  }
42
43  void print_token(Token* t) {
44      printf("> [%s]", t->class_name);
45      printf(" >>%s<<", t->str);
46      printf(" at (%ld, %ld), with size %ld\n", t->line, t->column
            , t->size);
47  }
48
49  /**
50   * This is a very dummy implementation for a search'n'insert
        operation on
51   * a 'set'.
52   */
53  void state_from_name(char* statename, State** st) {
54      int i, size;
55      char* pch;
56      // find a state that matchs, if so, return it within st
57      for (i = 0; i < _number_of_states; i++) {
58          if (strcmp(statename, state_table[i]->name) == 0) {
59              *st = state_table[i];
60              return;
61          }
62      }
```

```
63      // malloc size of State, here we do not care about freeing
            states,
64      // since the lex parser will run until the life span of the
            compiler run,
65      // the memory will be used until the end. No need to free it
            .
66      state_table[_number_of_states] = malloc(sizeof(State));
67      state_table[_number_of_states]->name = malloc(
68          sizeof(char) * (strlen(statename) + 1) // +1 for the \0
69      );
70
71      strcpy(state_table[_number_of_states]->name, statename);
72      // we should cut the '_' they are just different variations
            of the same class
73      pch = strrchr(statename, '_');
74      if (!pch) {
75          size = strlen(statename);
76      } else {
77          size = pch - statename;
78      }
79      state_table[_number_of_states]->class_name = malloc(
80          sizeof(char) * (size + 1)
81      );
82      strncpy(state_table[_number_of_states]->class_name,
            statename, size);
83      *st = state_table[_number_of_states++];
84  }
85
86  void add_mask_to_state(State** from, State** to, long* mask) {
87      (*from)->masks[(*from)->number_of_transitions] = mask;
88      (*from)->transitions[(*from)->number_of_transitions++] = *to
            ;
89  }
90
91  int lex_parser_read_char(FILE* f) {
92      char fromname[MAXLENGTHSTATESTR];
93      char toname[MAXLENGTHSTATESTR];
94      long *mask;
95      char sep;
```

```
96      char c;
97      long cod;
98      long maskterm, maskdepl;
99      int i;
100     State *from;
101     State *to;
102
103     long masktermsize = sizeof(long) * 8; // number of byts on a
            long
104
105     if (fscanf(f, " %c", &sep) == EOF || sep == EOF) {
106         return 0;
107     }
108     // complete mask of chars
109     mask = malloc(ENCODING_MAX_CHAR_NUM / (8));
110     for (i = 0; i < ENCODING_MAX_CHAR_NUM / (8 * sizeof(long));
            i++) {
111         // operator that means "all transitions" (in order to
                simulate the 'transductor')
112         mask[i] = (sep == '@')?(-1L):(0L);
113     }
114     // for each char different from sep, insert a transition
115     while (fscanf(f, "%c", &c) && c != sep && c != EOF) {
116         cod = (long) c;
117         maskterm = cod / masktermsize;
118         maskdepl = cod % masktermsize;
119         mask[maskterm] |= (1L<<maskdepl);
120     }
121     // origin state
122     fscanf(f, " %s", fromname);
123     state_from_name(fromname, &from);
124     // destiny state
125     fscanf(f, " %s", toname);
126     state_from_name(toname, &to);
127     add_mask_to_state(&from, &to, mask);
128     return 1;
129 }
130
131
```

```
132 void find_next_state_from_char(char c, State** from, State** to)
        {
133     long masktermsize = sizeof(long) * 8; // number of byts on a
            long
134     long cod, maskterm, maskdepl;
135     int i;
136     (*to) = NULL;
137     cod = (long) c;
138     maskterm = cod / masktermsize;
139     maskdepl = cod % masktermsize;
140     for (i = 0; i < (*from)->number_of_transitions; i++) {
141         // search for mathing out states.
142         if ((*from)->masks[i][maskterm] & (1L<<maskdepl)) {
143             (*to) = (*from)->transitions[i];
144             break;
145         }
146     }
147 }
148
149 int next_useful_token(FILE* f, Token** t) {
150     int res, i;
151
152     do {
153         res = next_token(f, t);
154     } while(
155         *t != NULL &&
156         res &&
157         strcmp((*t)->origin_state->class_name, "SPACE") == 0
158         // ignore SPACES
159     );
160
161     if (!res || *t == NULL){
162         return res; // if error or no token, return it to the
                caller.
163     }
164
165     if (strcmp((*t)->origin_state->class_name, "IDENT") == 0) {
166         for (i = 0; i < vkeywords_size; i++) {
```

```
167             // dummy search for keywords, this should become a
                    hashtable
168             // for the next project
169             if (strcmp((*t)->str, vkeywords[i]) == 0) {
170                 break;
171             }
172         }
173         if (i == vkeywords_size) {
174             (*t)->class_name = malloc(6 * sizeof(char));
175             strcpy((*t)->class_name, "IDENT");
176         } else {
177             (*t)->class_name = malloc(9 * sizeof(char));
178             // name it RESERVED in case it is
179             strcpy((*t)->class_name, "RESERVED");
180         }
181     } else {
182         (*t)->class_name = malloc(
183             (strlen((*t)->origin_state->class_name) + 1) *
                    sizeof(char)
184         );
185         strcpy((*t)->class_name, (*t)->origin_state->class_name)
                ;
186     }
187     // to be sure that this will not be used
188     (*t)->origin_state = NULL;
189     return res;
190 }
191
192 int next_token(FILE* f, Token** t) {
193     static State *current_state = NULL;
194     static long cline = 1;
195     static long ccolumn = 0;
196     static long line = 1;
197     static long column = 1;
198     static char tmpend = 1;
199     char next_c;
200
201     State* next_state;
202     // tmpend is static, if i read something that was
```

```
203     // EOF in the last step, this is the end and I should set t
            to null
204     if (tmpend == EOF) {
205         (*t) = NULL;
206         return 1;
207     }
208
209     if (current_state == NULL) {
210         // current_state is null, it means that this is
                initialization
211         // change it to Q0 and set the buffer to ""
212         state_from_name("Q0", &current_state);
213         buff_token_end = 0;
214         buff_token[0] = '\0';
215     }
216
217     do {
218         // get char, lookahead
219         tmpend = fscanf(f, "%c", &next_c);
220         if (next_c == '\n') { // column management
221             cline++;
222             ccolumn = 0;
223         } else {
224             ccolumn++;
225         }
226
227         next_state = NULL;
228         // let's see if there's a defined next state
229         find_next_state_from_char(next_c, &current_state, &
                next_state);
230         // if next state is Q0, it means that this is
                acceptation,
231         // we should stop, go to Q0 and reevaluate the
                transition.
232         //  Since the transductor have an empty transition to Q0
                , we are
233         //  obligated to do so.
234         if (next_state != NULL && strcmp(next_state->name, "Q0")
                == 0){
```

```
235             (*t) = malloc(sizeof(Token));
236             (*t)->str = malloc(sizeof(char) * (strlen(buff_token
                    ) + 1L));
237             strcpy((*t)->str, buff_token);
238             (*t)->line = line;
239             (*t)->column = column;
240             (*t)->origin_state = current_state;
241             (*t)->size = strlen(buff_token);
242             find_next_state_from_char(next_c, &next_state, &
                    current_state);
243             column = ccolumn;
244             line = cline;
245             // memorize next_c
246             buff_token[0] = next_c;
247             buff_token[1] = '\0';
248             buff_token_end = 1;
249             // no current_state but no end of file either, this
                    seams to be a
250             // problem.
251             if (current_state == NULL && tmpend != EOF) {
252                 fprintf(
253                         stderr,
254                         "buff_token (1): <%s>, error at line %ld
                                column %ld\n",
255                         buff_token,
256                         cline,
257                         ccolumn
258                 );
259                 return 0;
260             }
261             return 1;
262         } else {
263             buff_token[buff_token_end++] = next_c;
264             buff_token[buff_token_end] = '\0';
265         }
266         // no next state, raise error.
267         if (next_state == NULL) {
268             fprintf(
269                     stderr,
```

```
270                     "buff_token␣(2):␣<%s>,␣error␣at␣line␣%ld␣column␣
                            %ld\n",
271                     buff_token,
272                     cline,
273                     ccolumn
274                 );
275                 return 0;
276             }
277             current_state = next_state;
278         } while (tmpend != EOF);
279         (*t) = NULL;
280         return 1;
281 }
282
283 void initialize_lex() {
284         FILE *lex_file, *keywords_file;
285         vkeywords_size = 0;
286         _number_of_states = 0;
287
288         lex_file = fopen("./languagefiles/lang.lex", "r");
289         keywords_file = fopen("./languagefiles/keywords.txt", "r");
290
291         // parse the configuration file
292         while (lex_parser_read_char(lex_file)) {
293         }
294         // read keywords file
295         while (fscanf(keywords_file, "␣%s", buff_token) != EOF) {
296             vkeywords[vkeywords_size] = malloc(sizeof(char) * (
                    strlen(buff_token) + 1L));
297             strcpy(vkeywords[vkeywords_size++], buff_token);
298         }
299 }
```

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;

    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;
}
```