Bruno Duarte Barreto Borges
Fabio Oliveira de Abreu
Erik Kazuo Sugawara

Relatório

Florianópolis, SC

2021

Bruno Duarte Barreto Borges
Fabio Oliveira de Abreu
Erik Kazuo Sugawara

Relatório

Relatório sobre a Construção de um Compilador, feito pelos alunos: Bruno Duarte Barreto Borges, Fabio Oliveira de Abreu e Erik Kazuo Sugawara. Para obtenção da aprovação na disciplina Construção de Compiladores, ministrada pelo Prof. Dr. Alvaro Junio Pereira Franco.

Universidade Federal de Santa Catarina

Departamento de Informática e Estatística

INE5426 - Construção de Compiladores

Orientador: Alvaro Junio Pereira Franco

Florianópolis, SC

2021

Lista de abreviaturas e siglas

PLY	Python Lexx-Yacc
BNF	Bakus-Naur Form

Sumário

Introdução

Um compilador é um programa de sistema que traduz uma linguagem de alto nível para um programa equivalente em código de máquina para um processador. Sendo assim, um compilador não produz diretamente o código de máquina, mas sim um programa semanticamente equivalente na linguagem simbólica (assembly), que é traduzido para o programa em linguagem de máquina através de montadores[1].

Este trabalho, em sua primeira fase, tem como objetivo criar um compilador através de um analisador léxico e sintático para uma linguagem (AL). Para facilitar os estudos na criação do compilador, foi disponibilizada a linguagem "LCC-2021-2" derivada da gramática "CC-2021-2" na forma BNF(Backus-Naur Form), inspirada na gramática X++ de Delamaro [2].

As atividades contidas neste relatório foram feitas com a linguagem Python em conjunto da ferramenta PLY (Python Lex-Yacc)[3], uma implementação dos geradores de analisadores léxico e sintático (Lex-Yacc) para Python. Dessa forma, com o auxílio de ferramentas que permitem gerar analisadores, mostraremos o funcionamento do analisador léxico através de seu algoritmo e da sua aplicação com exemplos de programas em alto nível baseadas na gramática "CC-2021-2".

[1] <https://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node37.html>
[2] <http://conteudo.icmc.usp.br/pessoas/delamaro/SlidesCompiladores/CompiladoresFinal.pdf>
[3] <https://www.dabeaz.com/ply/>

Parte I

Ferramenta

1 Python Lex-Yacc

1.1 Sobre a ferramenta

O PLY é uma implementação pura em Python baseada nas ferramentas de construção de compiladores lex e yacc. Possui suporte para LARL(1) e mecanismos de validação, verificação de erros e diagnósticos. Sua primeira versão foi desenvolvida na Universidade de Chicago, para dar suporte à disciplina Introdução a Compiladores. A sua versão mais recente é o PLY-4.0, que requer o Python na versão 3.6 ou mais nova.

Ele consiste em dois módulos separados: o lex.py e o yacc.py. O módulo lex.py é utilizado para separar as entradas em texto do código em uma coleção de tokens baseadas em suas respectivas expressões regulares. O yacc.py é utilizado para reconhecer a sintaxe de uma linguagem baseada em uma gramática livre de contexto. Além disso, possui utilidades como verificação de erros, validação de gramática, suporte para produções vazias, tokens de erros e regras de precedência para resolver ambiguidades.

1.2 Lex

O Lex é utilizado para realizar a atribuição dos tokens com as entradas (*strings*). Os nomes dos tokens podem ser declarados de forma simples através de uma lista. Um token que é identificado se torna um objeto da classe LexToken.

```
import ply.lex as lex
import ply.yacc as yacc

# RESERVED WORDS
reserved = [
    'DEF',          # def
    'BREAK',        # break
    'FOR',          # for
    'IF',           # if
    'ELSE',         # else
    'INT',          # int
    'FLOAT',        # float
    'NEW',          # new
    'PRINT',        # print
    'READ',         # read
    'RETURN',       # return
    'STRING',       # string
]
```

Dada uma lista de tokens, podemos definir suas expressões regulares, que podem ser de forma simples ou complexa. Note que por definição da ferramenta, as expressões regulares são declaradas como uma variável que se inicia com "t_", seguida pelo nome do token definido na lista. Segue um exemplo das expressões regulares para tokens simples, como: *Assign, Greater Than, Less Than, Equals, Less Equal, Greater Equal, Not Equal, Plus, Minus, Multiply, Divide e Rem.*

```
t_ASSIGN = r'\='
t_GT = r'\>'
t_LT = r'\<'
t_EQ = r'\=='
t_LE = r'\<='
t_GE = r'\>='
t_NEQ = r'\!='
t_PLUS = r'\+'
t_MINUS = r'\-'
t_MULTIPLY = r'\*'
t_DIVIDE = r'\/'
t_REM = r'\%'
```

Para tokens que requerem maior complexidade para serem identificados, podemos declará-los através de funções que também se iniciam pelos caracteres "t_" seguidos pelo nome do token definido. Note que também podemos realizar o tratamento dos valores obtidos acessando o atributo *value* do objeto LexToken.

```
def t_float_constant(t):
    r'[+-]?\d+\.\d+([eE][+-]?\d+)?'
    t.value = float(t.value)
    return t

def t_int_constant(t):
    r'[+-]?\d+'
    t.value = int(t.value)
    return t

def t_string_constant(t):
    r'"[^"\n\r]*"'
    return t

def t_IDENT(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    return t
```

Com essa ferramenta também podemos fazer tratativas de leitura e identificação de erros. Ela também segue as definições de nomenclatura para tokens. Seguem alguns exemplos de tratamento de leitura para ignorar caracteres em branco, fim de linhas e caracteres ilegais.

```
t_ignore = r' ' # Ignore spaces between char.

def t_newline(t):
  r'\n+'
  t.lexer.lineno += len(t.value)

def t_error(t):
    errors.append("Illegal char %s in line %d, column %d" % (t.value[0],
                                            t.lexer.lineno, find_column(t))
                                           )
    t.lexer.skip(1)
```

Para demonstrar o funcionamento do analisador léxico, executamos o seguinte código. A saída é um LexToken, onde os atributos correspondem respectivamente ao identificador do token, símbolo, linha e coluna.

```
# Test it out
data = '''
3 + 4 * 10
'''

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    print(tok)
```

Saída :

```
LexToken(NUMBER,3,2,1)
LexToken(PLUS,'+',2,3)
LexToken(NUMBER,4,2,5)
LexToken(MULTIPLY,'*',2,7)
LexToken(NUMBER,10,2,10)
```

Parte II

Desenvolvimento

2 Metodologia

2.1 Extração dos tokens

Para obter os tokens, utilizamos a gramática "CC-2021-2" como base e dividimos cada um deles em um dos seguintes grupos: palavras reservadas, operadores, símbolos especiais, constantes e identificadores. Além disso, alteramos a gramática de modo que seja possível retornar valores com o token "*return*".

```
# RESERVED WORDS
reserved = [
    'DEF',          # def
    'BREAK',        # break
    'FOR',          # for
    'IF',           # if
    'ELSE',         # else
    'INT',          # int
    'FLOAT',        # float
    'NEW',          # new
    'PRINT',        # print
    'READ',         # read
    'RETURN',       # return
    'STRING',       # string
]
```

```
# OPERATORS
operators = [
    'ASSIGN',       # =
    'GT',           # >
    'LT',           # <
    'EQ',           # ==
    'LE',           # <=
    'GE',           # >=
    'NEQ',          # !=
    'PLUS',         # +
    'MINUS',        # -
    'MULTIPLY',     # *
    'DIVIDE',       # /
    'REM'           # %
]
```

```
# SPECIAL SYMBOLS
special = [
    'LPAREN',       # (
    'RPAREN',       # )
    'LBRACE',       # {
    'RBRACE',       # }
    'LBRACKET',     # [
    'RBRACKET',     # ]
    'SEMICOLON',    # ;
    'COMMA',        # ,
]
```

```
# CONSTANTS
constant = [
    'int_constant',
    'string_constant',
    'float_constant',
    'null_constant'
]
```

```
# IDENTIFIERS
identifiers = [
    'IDENT'
]
```

```
tokens = reserved + operators + special + constant + identifiers
```

2.2 Expressões Regulares

Após a identificação dos tokens, criamos a expressão regular de cada um deles.

```
t_ignore = r' ' # Ignore spaces between char.

def t_DEF(t):
    r'def'
    return t

def t_BREAK(t):
    r'break'
    return t

def t_FOR(t):
    r'for'
    return t
```

```
def t_IF(t):
    r'if'
    return t

def t_ELSE(t):
    r'else'
    return t

def t_NEW(t):
    r'new'
    return t

def t_PRINT(t):
    r'print'
    return t

def t_READ(t):
    r'read'
    return t

def t_RETURN(t):
    r'return'
    return t

def t_STRING(t):
    r'string'
    return t

def t_INT(t):
    r'int'
    return t

def t_FLOAT(t):
    r'float'
    return t
```

```
def t_null_constant(t):
    r'null'
    return t

def t_float_constant(t):
    r'[+-]?\d+\.\d+([eE][+-]?\d+)?'
    t.value = float(t.value)
    return t
```

```
def t_int_constant(t):
    r'[+-]?\d+'
    t.value = int(t.value)
    return t

def t_string_constant(t):
    r'"[^"\n\r]*"'
    return t

def t_IDENT(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    return t

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

errors = []

def t_error(t):
    errors.append("Illegal char %s in line %d, column %d" % (t.value[0],
                                        t.lexer.lineno, find_column(t))
                                       )
    t.lexer.skip(1)
```

Além da criação das expressões regulares, também foi criado uma função que imprime a tabela de símbolos de forma elegante no terminal.

```
def print_table(lexer):
    pattern = "{:^25} | {:^60} | {:^7} | {:^7}"
    print("\033[4m" + pattern.format("TOKEN", "VALUE", "LINE", "COLUMN")
                                        + "\033[0m")
    while True:
        tok = lexer.token()
        if not tok:
            break
        print(pattern.format(tok.type, tok.value, tok.lineno,
                                            find_column(tok)))

    for e in errors:
        print(e)
```

2.3 Analisador Léxico

Com os tokens e expressões regulares definidos, executamos o analisador léxico e obtemos a sua tabela de símbolos formatada através da função "print_table".

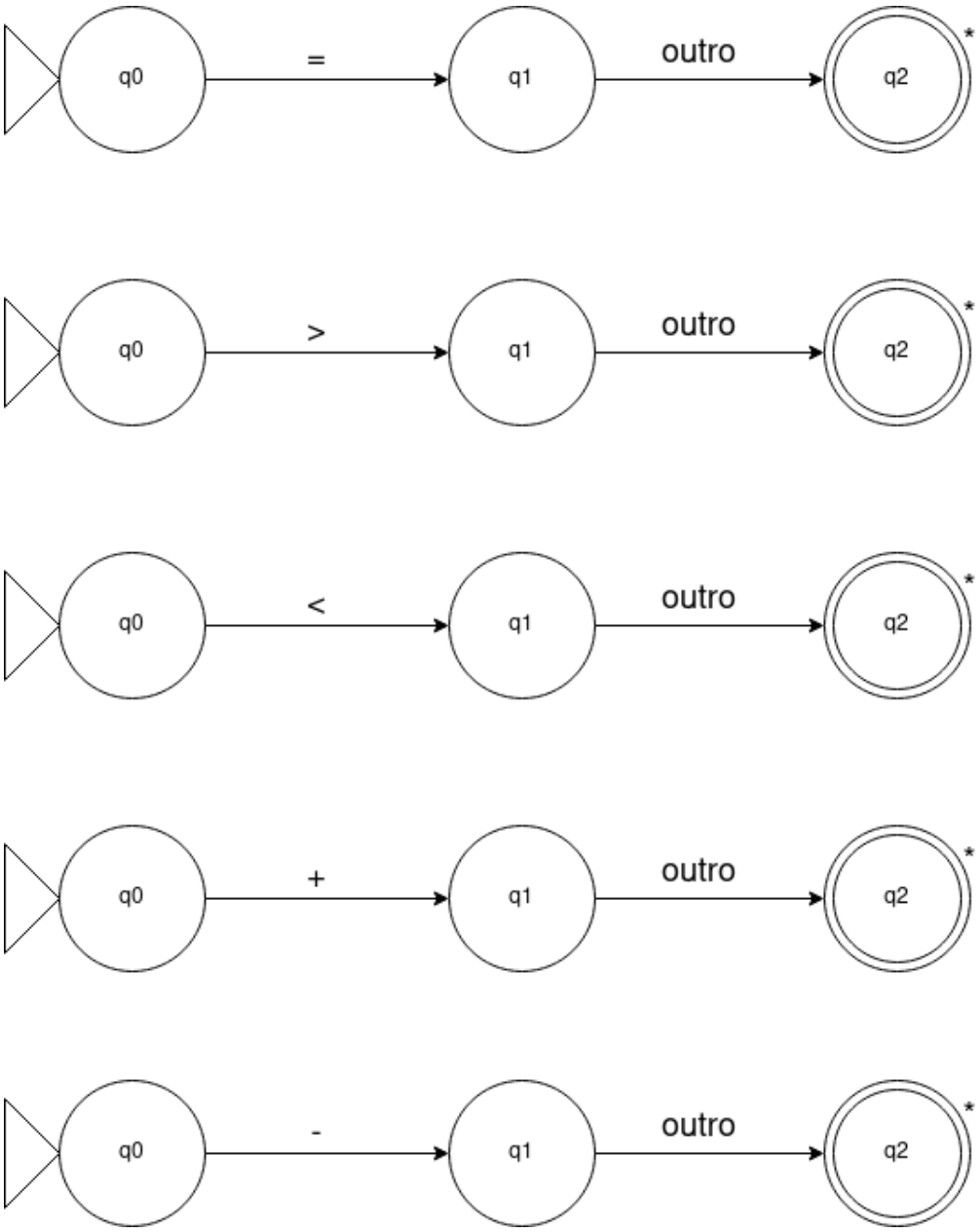
```
$ make run file='tmp/lil_example.lcc'
```

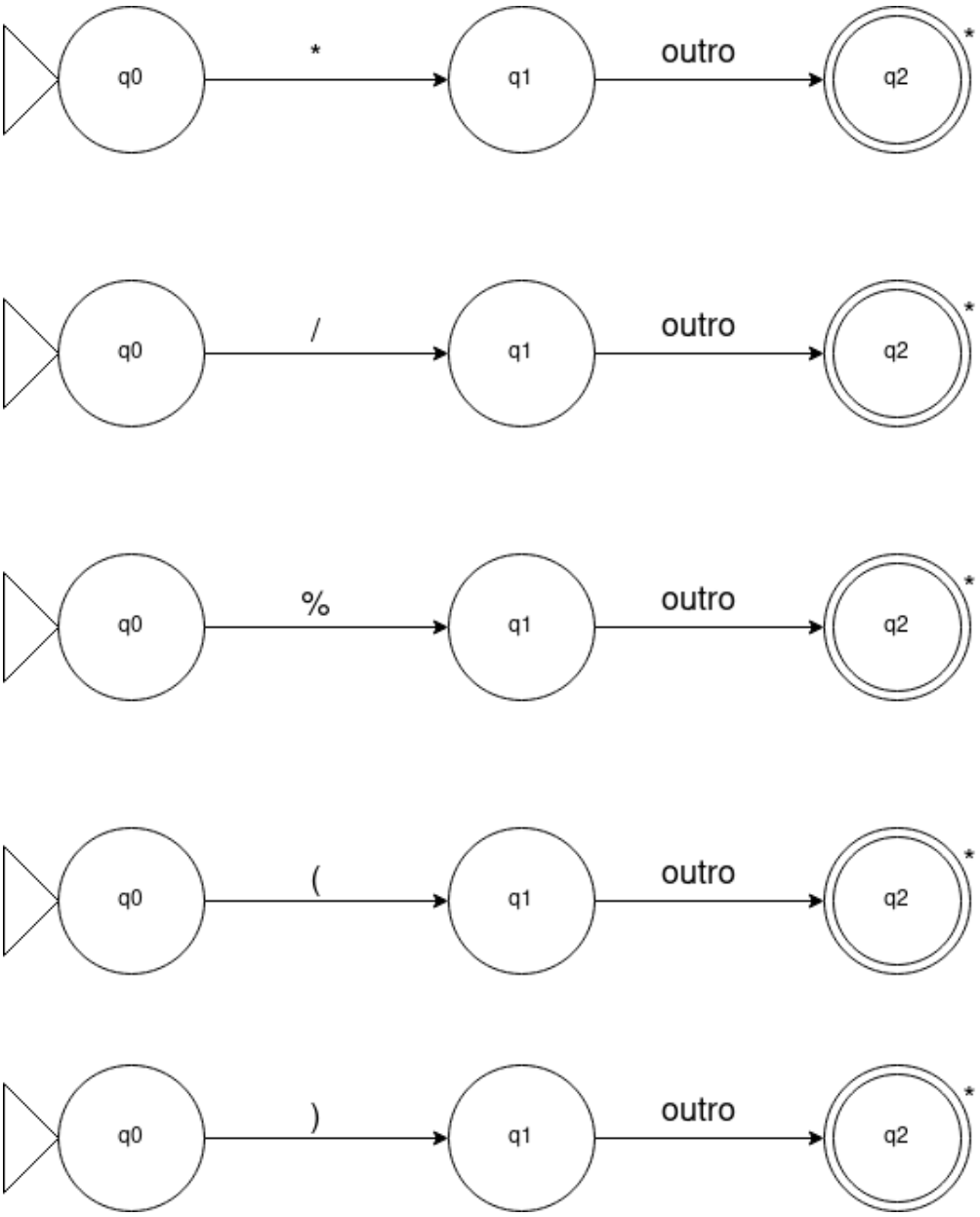
TOKEN	VALUE	LINE	COLUMN
DEF	def	1	1
IDENT	hello_world	1	5
LPAREN	(	1	16
RPAREN	)	1	17
LBRACE	{	1	19
PRINT	print	2	5
string_constant	"hello␣world"	2	11
SEMICOLON	;	2	24
RBRACE	}	3	1
INT	int	5	1
IDENT	x	5	5
SEMICOLON	;	5	6
IDENT	x	6	1
ASSIGN	=	6	3
int_constant	10	6	5
SEMICOLON	;	6	7
IF	**if**	8	1
LPAREN	(	8	4
IDENT	x	8	5
GT	>	8	7
int_constant	30	8	9
RPAREN	)	8	11
LBRACE	{	8	13
IDENT	hello_world	9	5
LPAREN	(	9	16
RPAREN	)	9	17
SEMICOLON	;	9	18
RBRACE	}	10	1
ELSE	**else**	10	3
LBRACE	{	10	8
PRINT	print	11	5
string_constant	"erro"	11	11
SEMICOLON	;	11	17
RBRACE	}	12	1

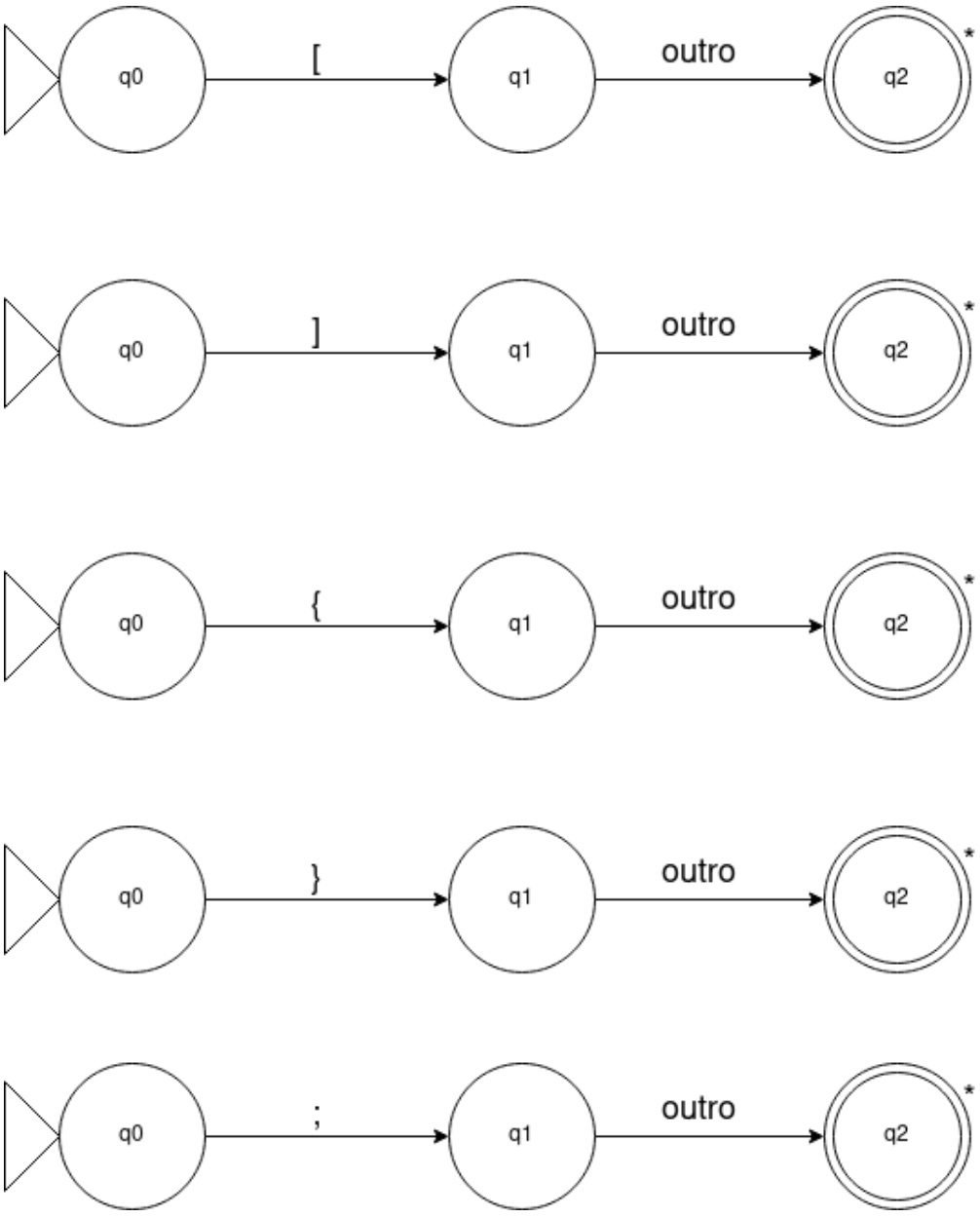
Parte III

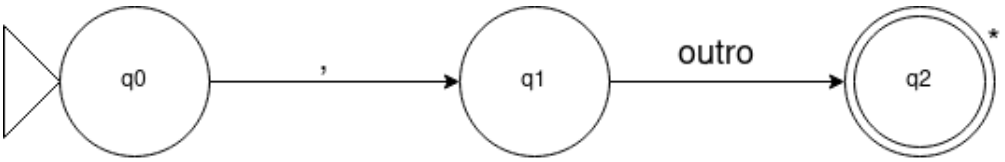
Diagramas de Transição

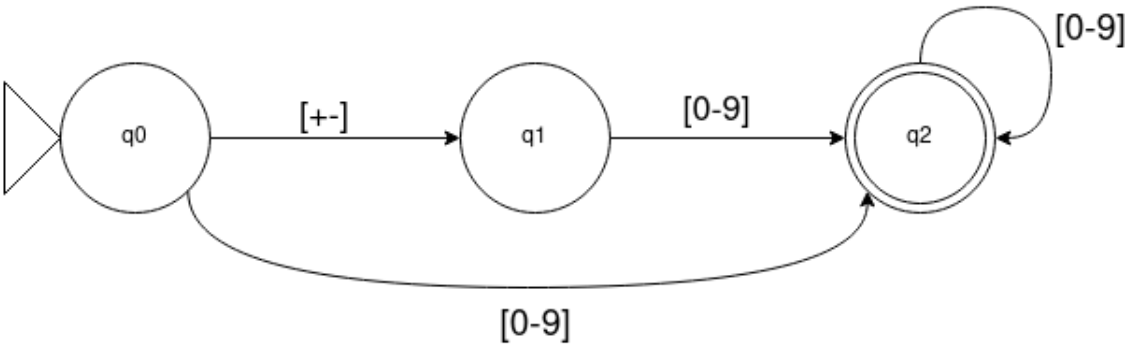
3 Diagrama de Transição

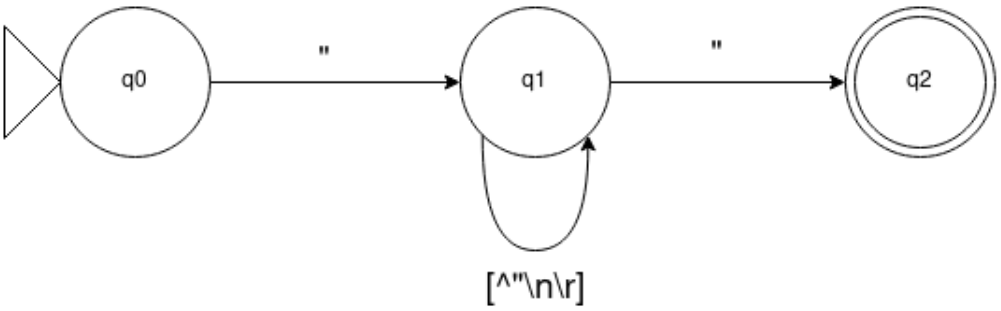


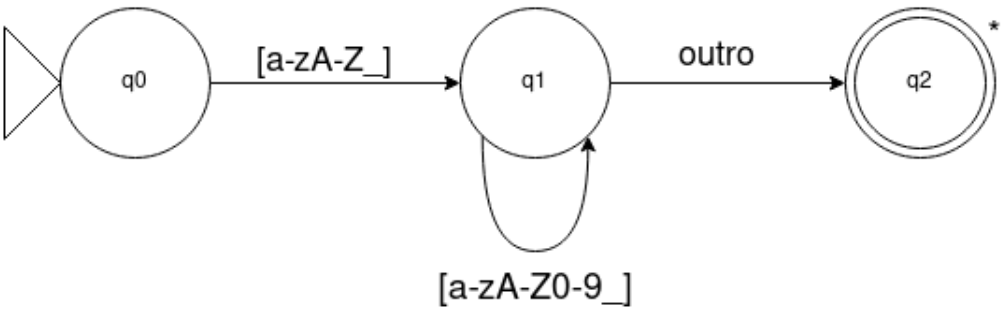

q0
*
q1
outro
q2
*
q0
/
q1
outro
q2
*
q0
%
q1
outro
q2
*
q0
(
q1
outro
q2
*
q0
)
q1
outro
q2
*

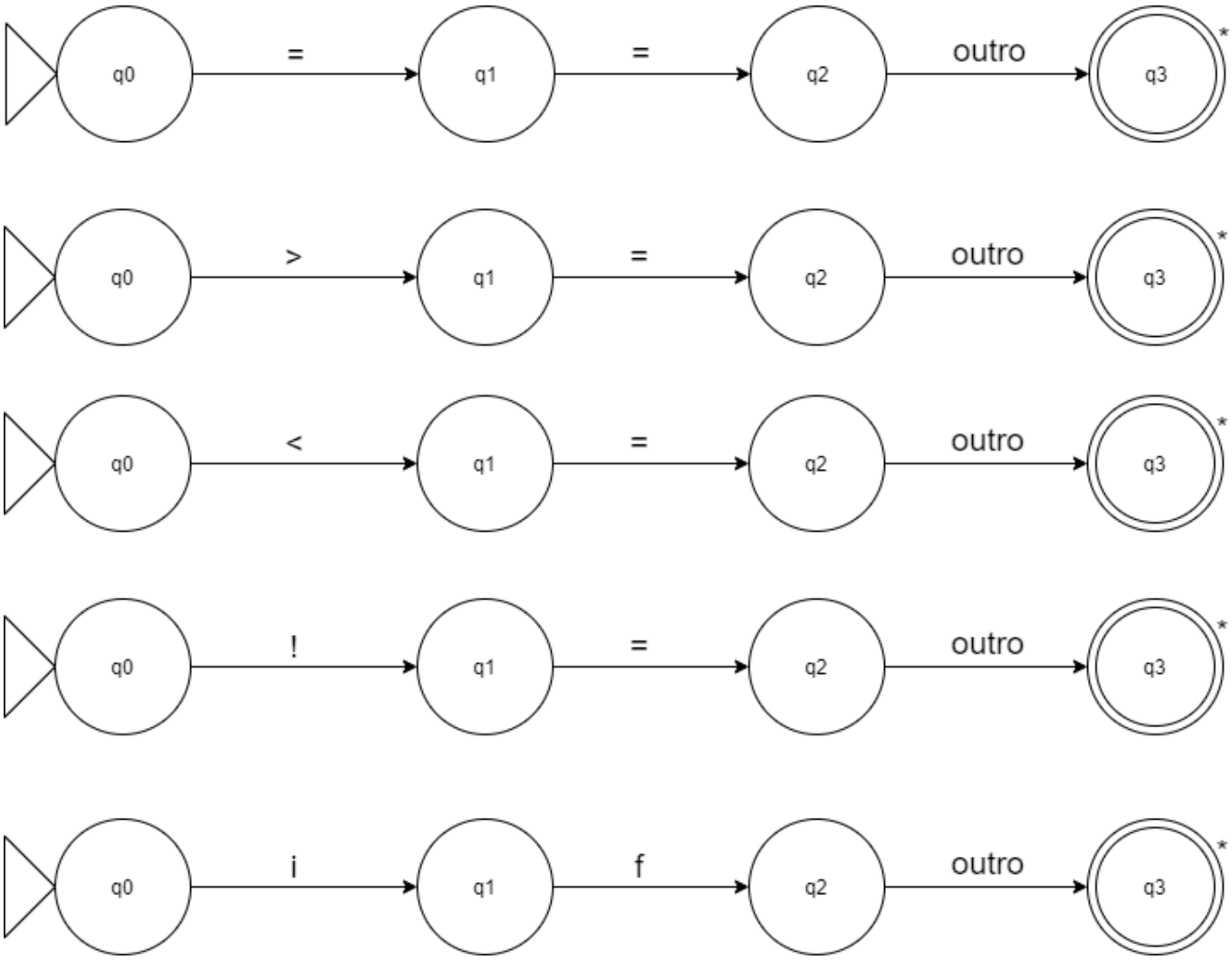

q0
[
q1
outro
q2
*
q0
]
q1
outro
q2
*
q0
{
q1
outro
q2
*
q0
}
q1
outro
q2
*
q0
;
q1
outro
q2
*

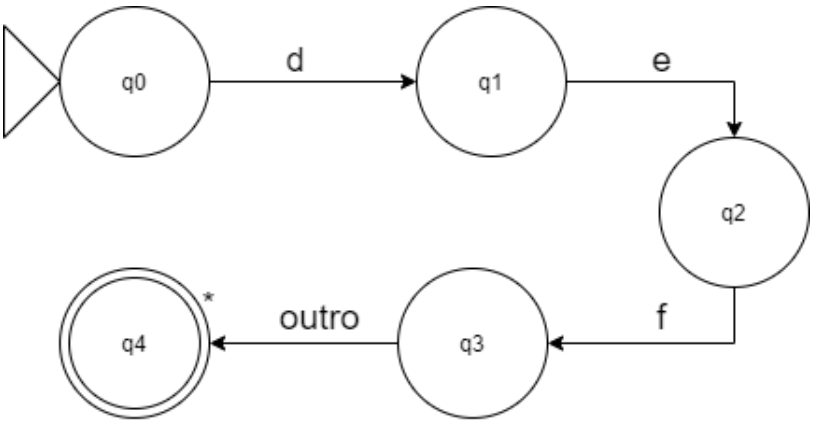

q0
'
q1
outro
q2
*

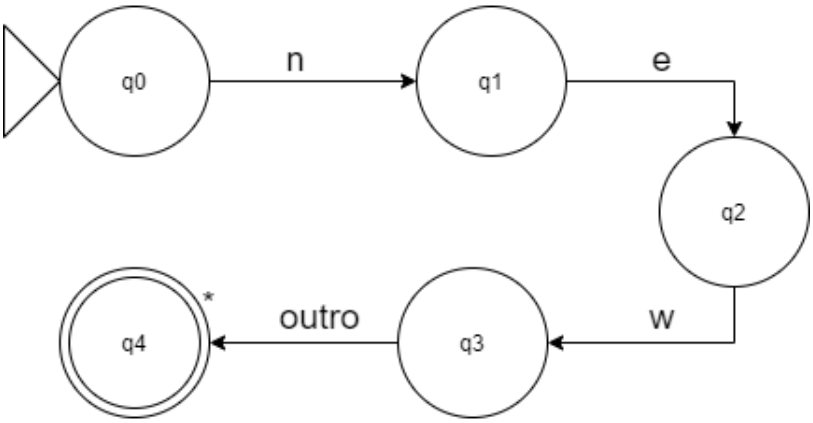

q0
[+-]
q1
[0-9]
q2
[0-9]
[0-9]

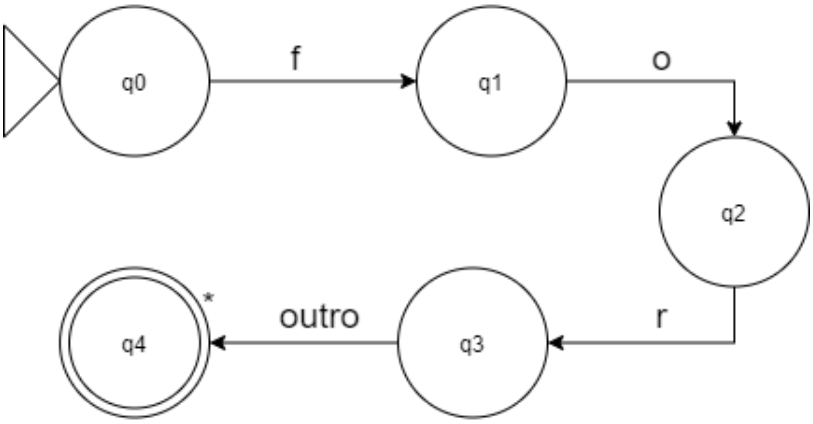

q0
"
q1
"
q2
[^"\n\r]

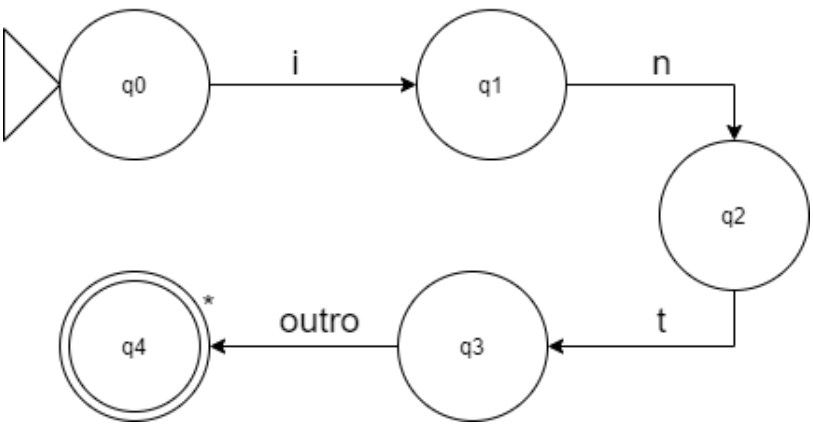

q0
[a-zA-Z_]
q1
outro
q2
*
[a-zA-Z0-9_]

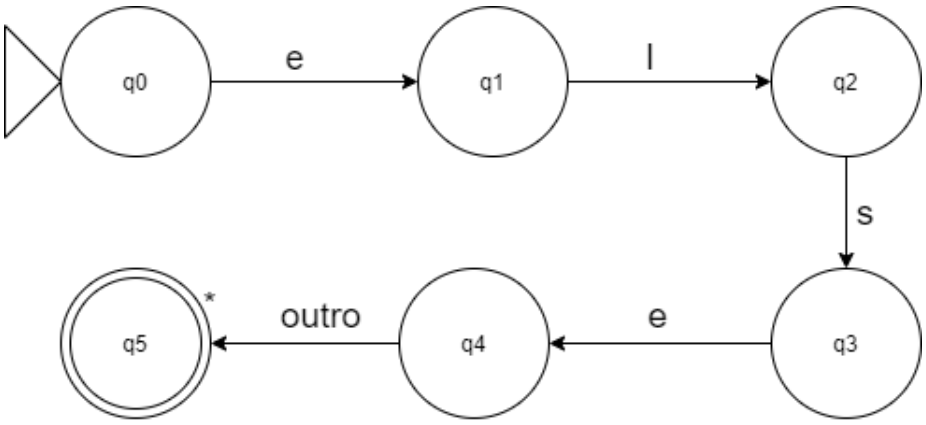

q0
=
q1
=
q2
outro
q3
*
q0
>
q1
=
q2
outro
q3
*
q0
<
q1
=
q2
outro
q3
*
q0
!
q1
=
q2
outro
q3
*
q0
i
q1
f
q2
outro
q3
*

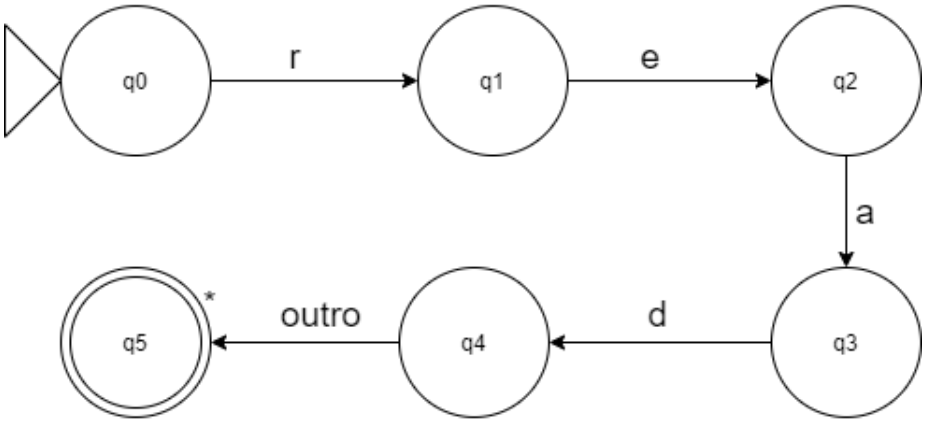

q0
d
q1
e
q2
f
q3
outro
q4
*

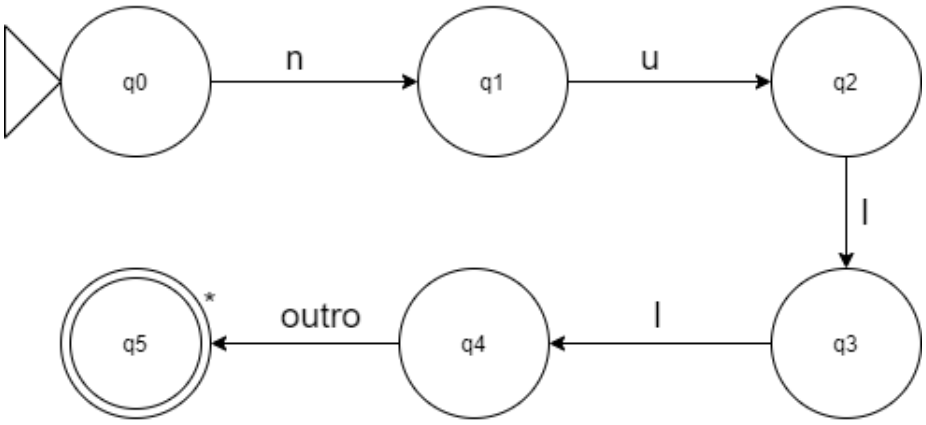

q0
n
q1
e
q2
w
q3
outro
q4
*

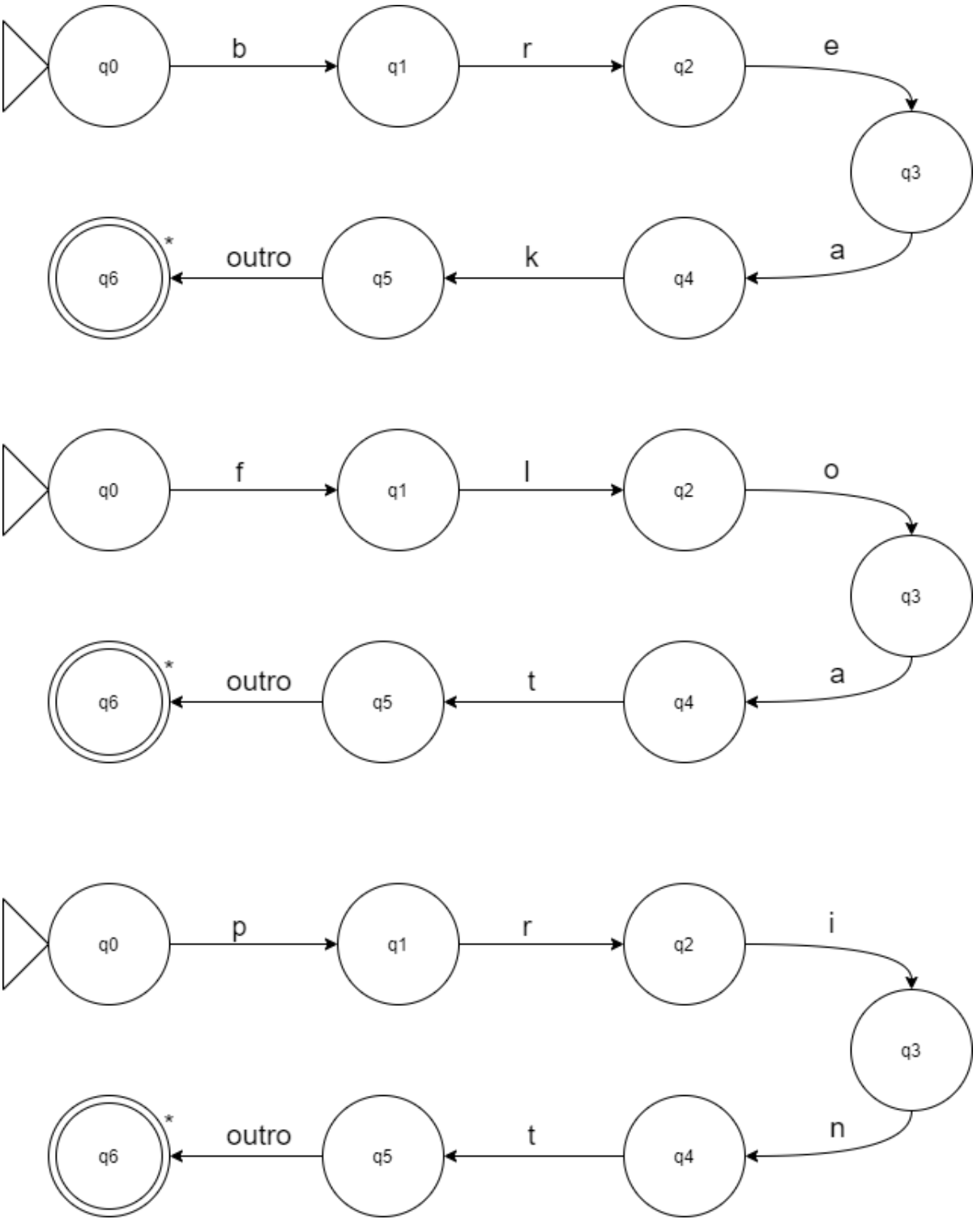

q0
f
q1
o
q2
r
q3
outro
q4
*

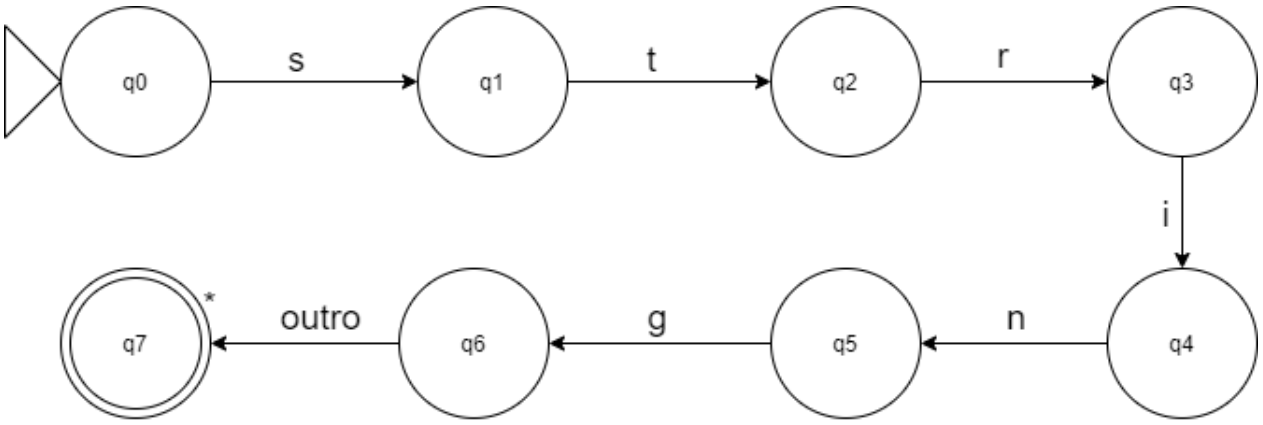

q0
i
q1
n
q2
t
q3
outro
q4
*

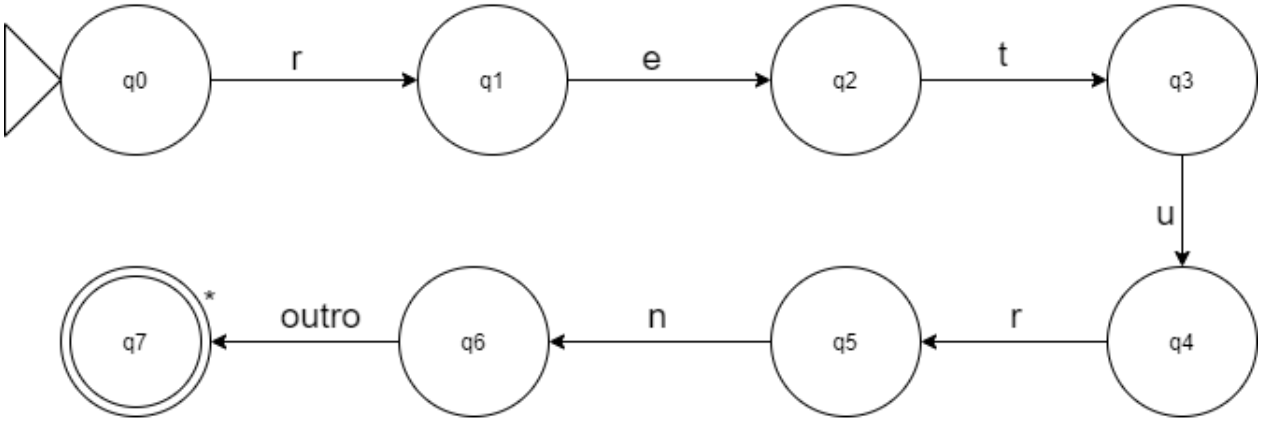

q0
e
q1
l
q2
s
q3
e
q4
outro
q5
*

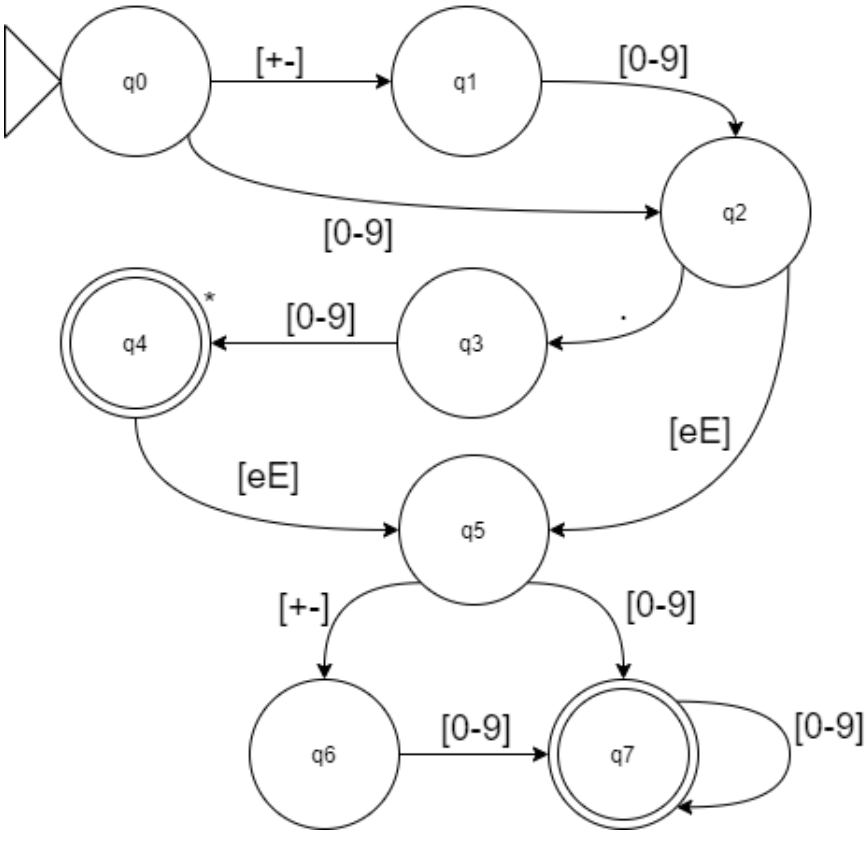

q0
r
q1
e
q2
a
q3
d
q4
outro
q5
*


q0
n
q1
u
q2
l
q3
l
q4
outro
q5
*


q0
b
q1
r
q2
e
q3
a
q4
k
q5
outro
q6
*
q0
f
q1
l
q2
o
q3
a
q4
t
q5
outro
q6
*
q0
p
q1
r
q2
i
q3
n
q4
t
q5
outro
q6
*


q0
s
q1
t
q2
r
q3
i
q4
n
q5
g
q6
outro
q7
*


q0
r
q1
e
q2
t
q3
u
q4
r
q5
n
q6
outro
q7
*


q0
[+-]
q1
[0-9]
q2
[0-9]
.
q3
[0-9]
q4
*
[eE]
[eE]
q5
[+-]
[0-9]
q6
[0-9]
q7
[0-9]

Conclusão

Os compiladores traduzem o código fonte de uma linguagem de programação de alto nível para uma linguagem de programação de baixo nível. Sem eles a tarefa de programar seria um trabalho extremamente lento e difícil. São nichos específicos e muito raros os casos em que desenvolvemos aplicações feitas diretamente em Assembly. Nos dias atuais, quase todas linguagens possui o seu compilador, facilitando e aumentando a efetividade dos programadores. Dito isto, com esse trabalho, foi possível entender a primeira parte da construção de um compilador, ou seja, do analisador léxico. Utilizando a ferramenta PLY, entendemos os princípios utilizados para se construir um compilador de uma linguagem qualquer através da sua aplicação prática. Além disso, com o Diagrama de Transição foi possível entender como ocorre a leitura das palavras e como são identificado os tokens. Portanto, ao realizar este trabalho, ficou evidente como criar um compilador através de ferramentas geradores de analisadores léxicos e sintáticos e dos estudos e pesquisas sobre compiladores.