INSTITUTO POLITÉCNICO
DO CÁVADO E DO AVE

Escola superior de Tecnologia

Processamento de linguagens

Autores:
Diogo Bernardo ***21144***
João Ribeiro ***23795***
José Fangueiro ***23515***

Docentes:
Prof. Óscar Ribeiro

Submissão do projeto para
o curso de Engenharia de Sistemas Informáticos

ESI

April 14, 2023

IPCA

Resumo

Engenharia de Sistemas Informáticos

Processamento de linguagens

por Diogo Bernardo **21144**
João Ribeiro **23795**
José Fangueiro **23515**

Trabalho em cargo da disciplina de Processamento de linguagens.

Contents

Chapter 1

Linguagens Regulares – Análise Léxica (Exercício 1)

1.1 Definição da expressão regular correspondente

$$\boldsymbol{R} = (\varepsilon \mid + \mid -)(\mathbf{0}\ldots\mathbf{9})^*((\varepsilon \mid \cdot(\mathbf{0}\ldots\mathbf{9})^+)(\varepsilon \mid (\boldsymbol{e} \mid \boldsymbol{E})(\varepsilon \mid + \mid -)(\mathbf{0}\ldots\mathbf{9})^+))$$

A expressão regular pretendida é usada para identificar números de virgula flutuante:

1. $(\varepsilon \mid + \mid -)$: Esta parte é opcional, um sinal de mais (+) ou um sinal de menos (-). Isto corresponde ao sinal do número, caso omitida é assumido o valor positivo.
2. $(0\ldots9)^*$: Esta parte corresponde a uma sequência de zero ou mais dígitos (0 a 9). Esta sequência forma a parte inteira do número, e pode ser omitida para considerar os casos ex:'.23'.
3. $((\varepsilon \mid \cdot(0\ldots9)^+)(\varepsilon \mid (e \mid E)(\varepsilon \mid + \mid -)(0\ldots9)^+))$: Esta parte descreve a parte fracionária e o expoente do número em notação científica.
   (a) $(\varepsilon \mid \cdot(0\ldots9)^+)$: Esta subexpressão é opcional, um ponto decimal (.) seguido de um ou mais dígitos (0 a 9). Corresponde à parte fracionária do número.
   (b) $(\varepsilon \mid (e \mid E)(\varepsilon \mid + \mid -)(0\ldots9)^+)$: Esta subexpressão descreve o expoente opcional em notação científica. Descreve a letra 'e' ou 'E' seguida por um sinal opcional (+ ou -) e uma sequência de um ou mais dígitos (0 a 9).

1.1.1 Definição da expressão regular correspondente na notação de programação

```
R = ^([+-]?)(\d*)((\.\d+)?([eE][+-]?\d+)?|\.\d+([eE][+-]?\d+)?$
```

Divisão da expressão:

1. `^[+-]?`: O `^` indica o início da string. `[+-]?` verifica se há um sinal de mais (+) ou menos (-) opcional no início do número.
2. `\d*`: Verifica dígitos (0-9), é opcional para na expressão ser aceite (ex:'.23').
3. `((\.\d+)?([eE][+-]?\d+)?)?`: Esta parte é um grupo opcional que verifica duas coisas:
   (a) `(\.\d+)?`: Um ponto decimal seguido de um ou mais dígitos, grupo opcional.

   (b) `([eE][+-]?\d+)?`: Um "e" ou "E" (para representar a notação científica) seguido de um sinal opcional de mais (+) ou menos (-) e um ou mais dígitos. Esta parte também é opcional.

4. `|\.\d([eE][+-]?+)?|+`: Esta parte verifica números que começam com um ponto decimal:

   (a) `\.\d+`: Um ponto decimal seguido de um ou mais dígitos.

   (b) `([eE][+-]?\d+)?`: Um "e" ou "E" (para representar a notação científica) seguido de um sinal opcional de mais (+) ou menos (-) e um ou mais dígitos. Esta parte também é opcional.

5. `$`: Indica o fim da string.

1.2 Calculo do autómato determinista que implementa o reconhecimento da expressão regular em a.

1.2.1 Representação da expressão regular como vários autómatos finitos não deterministas.

Sinal opcional:

Autómato: $A_1 = (Q, \Sigma, \delta, q_0, F)$

Expressão regular: $R = (+ \mid - \mid \epsilon)$

Estados: $Q = \{s_0, s_1\}$

Alfabeto: $\Sigma = \{+, -\}$

Função de transição: $\delta : Q \times \Sigma \to Q$

Estado	Transições	
s_0	$+$	s_1
	$-$	s_1
	ε	s_1
s_1		

Estado inicial: $q_0 = s_0$

Estados finais: $F = \{s_1\}$

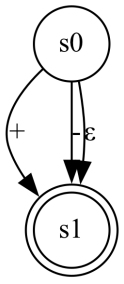

Figure 1.1: AFND 1

Parte inteira:

Autómato: $A_2 = (Q, \Sigma, \delta, q_0, F)$

Expressão regular: $R = (0 \dots 9)^*$

Estados: $Q = \{s_2, s_3\}$

Alfabeto: $\Sigma = \{(0 \dots 9)\}$

Função de transição: $\delta : Q \times \Sigma \to Q$

Estado	**Transições**	
s_2	$(0 \dots 9)$	s_2
	ε	s_3
s_3		

Estado inicial: $q_0 = s_2$

Estados finais: $F = \{s_3\}$

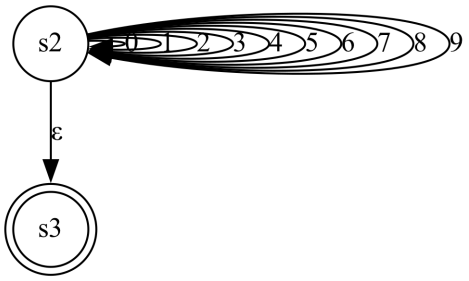

Figure 1.2: AFND 2

Parte decimal

Autómato: $A_3 = (Q, \Sigma, \delta, q_0, F)$

Expressão regular: $R = (\varepsilon \mid \cdot(0 \dots 9)^+)$

Estados: $Q = \{s_4, s_5, s_6\}$

Alfabeto: $\Sigma = \{(0 \dots 9).\}$

Função de transição: $\delta : Q \times \Sigma \to Q$

Estado	**Transições**	
s_4	$.$	s_5
	ε	s_6
s_5	$(0 \dots 9)$	s_5
s_6		

Estado inicial: $q_0 = s_4$

Estados finais: $F = \{s_5, s_6\}$

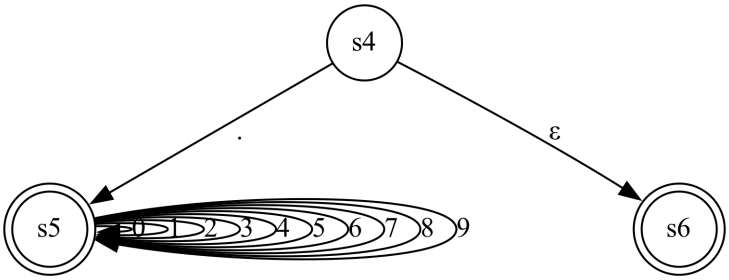

FIGURE 1.3: AFND 3

Parte exponencial:

Autómato: $A_4 = (Q, \Sigma, \delta, q_0, F)$

Expressão regular: $R = (\varepsilon \mid (e \mid E)(\varepsilon \mid + \mid -)(0 \ldots 9)^+)$

Estados: $Q = \{s_7, s_8, s_9, s_{10}\}$

Alfabeto: $\Sigma = \{(0 \ldots 9) + -Ee\}$

Função de transição: $\delta : Q \times \Sigma \to Q$

Estado	**Transições**	
s_7	E, e	s_8
	ε	s_{10}
s_8	$+, -, \varepsilon$	s_9
s_9	$(0 \ldots 9)$	s_9
s_{10}		

Estado inicial: $q_0 = s_7$

Estados finais: $F = \{s_9, s_{10}\}$

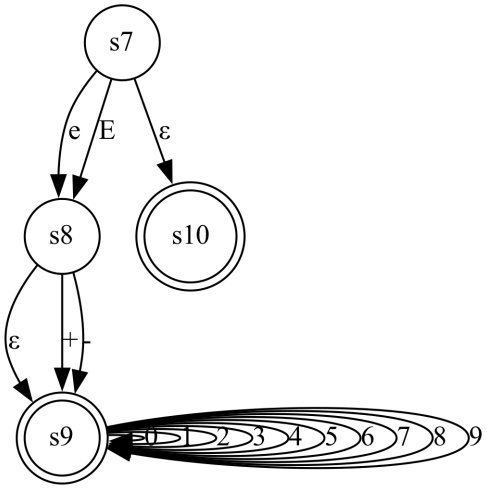

FIGURE 1.4: AFND 4

1.2.2 Autómato fínito não determinista obtido.

Autómato: $A_{nd} = (Q, V, \delta, q_0, F)$

Expressão: $R = (\varepsilon \mid + \mid -)(0 \ldots 9)^*((\varepsilon \mid \cdot(0 \ldots 9)^+)(\varepsilon \mid (e \mid E)(\varepsilon \mid + \mid -)(0 \ldots 9)^+))$

Estados: $Q = \{n_0, n_1, n_2, \ldots, n_{10}\}$

Alfabeto: $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, e, E, +, -, .\}$

Função de transição: $\delta : Q \times \Sigma \to Q$

Estados	**Transições**	
n_0	$+$	n_1
	$-$	n_1
	ε	n_1
n_1	$0 \ldots 9$	n_2
	ε	n_3
n_2	$0 \ldots 9$	n_2
	ε	n_4
n_3	ε	n_5
	$.$	n_6
n_4	ε	n_{10}
	e	n_7
	E	n_7
n_5	$0 \ldots 9$	n_5
n_6	ε	n_{10}
	e	n_7
	E	n_7
n_7	ε	n_8
	$+$	n_8
	$-$	n_8
n_8	$0 \ldots 9$	n_9
n_9	$0 \ldots 9$	n_9
n_{10}		

Estados iniciais: $q_0 = n_0$

Estados finais: $F = \{n3, n5, n9\}$

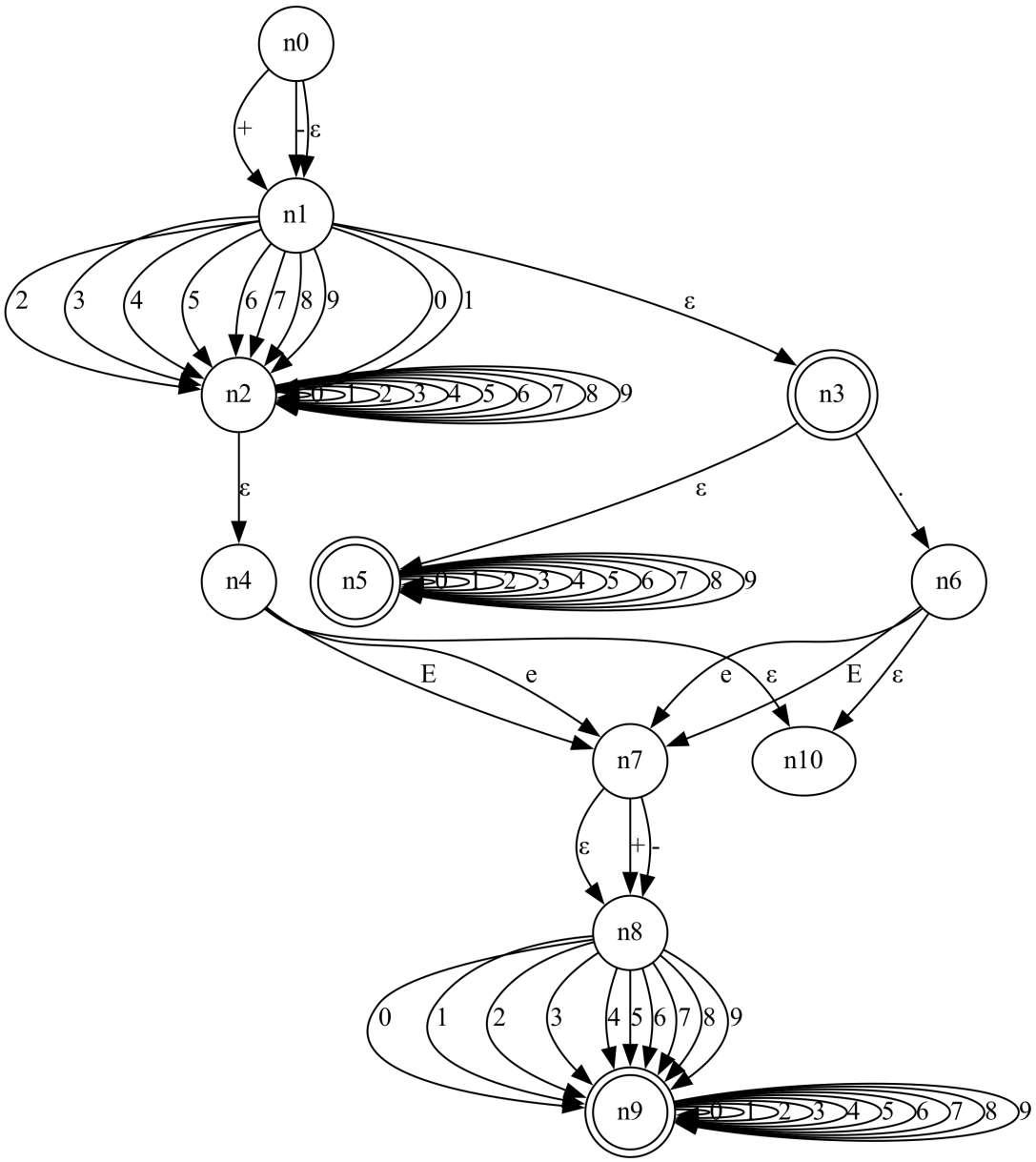

Figure 1.5: AFND

1.2.3 Conversão AFND -> AFD

Estados combinados não deterministas	**Estado determinista**	**Transições**
n0, n1	q0	+, - → n1 0-9 → n2 . → n3
n1	q1	0-9 → n2 . → n3
n2	q2	0-9 → n2 e, E → n4, n7, n10
n3	q3	0-9 → n5, n6
n4, n7, n10	q4	+, - → n8 0-9 → n5
n5, n6	q6	0-9 → n6
n5	q5	0-9 → n5
n6	—	0-9 → n6 e, E → n4, n7, n10
n8	—	0-9 → n9
n9	—	0-9 → n9

Table 1.1: AFND -> AFD

n6,n8,n9 têm alguma redundância, não representam diretamente um estado determinista.

Autómato finito determinista obtido:

Autómato: $A = (Q, V, \delta, q_0, F)$

Expressão: $R = (\varepsilon \mid + \mid -)(0 \ldots 9)^*((\varepsilon \mid \cdot(0 \ldots 9)^+)(\varepsilon \mid (e \mid E)(\varepsilon \mid + \mid -)(0 \ldots 9)^+))$

Estados: $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$

Alfabeto: $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, e, E, +, -, .\}$

Função de transição: $\delta : Q \times \Sigma \to Q$

Estado	**Transições**	
q_0	$+, -$	q_1
	$0 \ldots 9$	q_2
	$.$	q_3
q_1	$0 \ldots 9$	q_2
	$.$	q_3
q_2	$0 \ldots 9$	q_2
	$.$	q_3
	e, E	q_4
q_3	$0 \ldots 9$	q_6
q_4	$+, -$	q_5
	$0 \ldots 9$	q_5
q_5	$0 \ldots 9$	q_5
q_6	$0 \ldots 9$	q_6
	e, E	q_4

Estados iniciais: $q_0 = q_0$

Estados finais: $F = \{q_2, q_6, q_5\}$

1.3 Definição e teste da função de transição correspondente

Estado	Entrada	Próxima entrada
q0	+	q1
	-	q1
	0-9	q2
	.	q3
q1	0-9	q2
	.	q3
q2	0-9	q2
	.	q3
	e	q4
	E	q4
q3	0-9	q6
q4	+	q5
	-	q5
	0-9	q5
q5	0-9	q5
q6	0-9	q6
	e	q4
	E	q4

Table 1.2: Função de transição AFD

1.3.1 Algoritmo em python.

```
V = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "+", "-", ".", "e", "E"}
Q = {"q0", "q1", "q2", "q3", "q4", "q5", "q6"}
tt = {
   "q0": {
      "+": "q1",
      "-": "q1",
      "0": "q2",
      "1": "q2",
      "2": "q2",
      "3": "q2",
      "4": "q2",
      "5": "q2",
      "6": "q2",
      "7": "q2",
      "8": "q2",
      "9": "q2",
      ".": "q3",
   },
   "q1": {
      "0": "q2",
      "1": "q2",
      "2": "q2",
      "3": "q2",
      "4": "q2",
```

```
      "5": "q2",
      "6": "q2",
      "7": "q2",
      "8": "q2",
      "9": "q2",
      ".": "q3",
   },
   "q2": {
      "0": "q2",
      "1": "q2",
      "2": "q2",
      "3": "q2",
      "4": "q2",
      "5": "q2",
      "6": "q2",
      "7": "q2",
      "8": "q2",
      "9": "q2",
      ".": "q3",
      "e": "q4",
      "E": "q4",
   },
   "q3": {
      "0": "q6",
      "1": "q6",
      "2": "q6",
      "3": "q6",
      "4": "q6",
      "5": "q6",
      "6": "q6",
      "7": "q6",
      "8": "q6",
      "9": "q6",
   },
   "q4": {
      "+": "q5",
      "-": "q5",
      "0": "q5",
      "1": "q5",
      "2": "q5",
      "3": "q5",
      "4": "q5",
      "5": "q5",
      "6": "q5",
      "7": "q5",
      "8": "q5",
      "9": "q5",
   },
   "q5": {
      "0": "q5",
      "1": "q5",
      "2": "q5",
      "3": "q5",
      "4": "q5",
      "5": "q5",
      "6": "q5",
      "7": "q5",
```

```
      "8": "q5",
      "9": "q5",
   },
   "q6": {
      "0": "q6",
      "1": "q6",
      "2": "q6",
      "3": "q6",
      "4": "q6",
      "5": "q6",
      "6": "q6",
      "7": "q6",
      "8": "q6",
      "9": "q6",
      "e": "q4",
      "E": "q4",
   },
}

q0 = "q0"
F = {"q2", "q6", "q5"}


def is_recognized(string):
   """ Determines if a given string represents a valid float. """
   current_state = q0
   for char in string:
      current_state = tt.get(current_state, {}).get(char, None)
      if current_state is None:
         return False
   return current_state in F


def cast_to_float(test_cases):
   """ Cast the string a valid floating point number and print to stdout """
   print("\nConverting to float!")
   for string, expected in test_cases:
      if is_recognized(string) == expected and expected:
         float_repr = "{:,.10f}".format(float(string))
         print(f"{string.ljust(10)}: {float_repr}")


test_cases = [
   ("123", True),
   ("-123", True),
   ("+123", True),
   ("123.45", True),
   (".45", True),
   ("-123.45", True),
   ("+123.45", True),
   ("1.23e-4", True),
   ("-1.23E4", True),
   ("1.23e+4", True),
   ("1.23E-4", True),
   ("1.23E4", True),
   ("2E2", True),
   ("2e2", True),
```

```
    ("2e2.2", False),
    ("2.", False),
    ("abc", False),
    ("123a", False),
    ("--123", False),
    ("123.45.67", False),
    ("123.45.67", False),
    ("123.45a", False),
    ("123a2", False),
    ("123e2.2", False),
    ("123 ", False),
    (" 123 ", False),
]


def test():
    """ Unit testing for the automaton """
    matched_tests = 0
    for string, expected in test_cases:
        result = is_recognized(string)
        padded_string = f"'{string}'".ljust(11)
        if result == expected:
            matched_tests += 1
        print(
            f"str: {padded_string} is a float? expected({str(expected).ljust(5)}) got -> {result}"
        )
    print(f"\nMatched {matched_tests} out of {len(test_cases)} test cases!")


test()

cast_to_float(test_cases)
```

1.3.2 Teste do algoritmo em python:

```
str: '123'       is a float? expected(True ) got -> True
str: '-123'      is a float? expected(True ) got -> True
str: '+123'      is a float? expected(True ) got -> True
str: '123.45'    is a float? expected(True ) got -> True
str: '.45'       is a float? expected(True ) got -> True
str: '-123.45'   is a float? expected(True ) got -> True
str: '+123.45'   is a float? expected(True ) got -> True
str: '1.23e-4'   is a float? expected(True ) got -> True
str: '-1.23E4'   is a float? expected(True ) got -> True
str: '1.23e+4'   is a float? expected(True ) got -> True
str: '1.23E-4'   is a float? expected(True ) got -> True
str: '1.23E4'    is a float? expected(True ) got -> True
str: '2E2'       is a float? expected(True ) got -> True
str: '2e2'       is a float? expected(True ) got -> True
str: '2e2.2'     is a float? expected(False) got -> False
str: '2.'        is a float? expected(False) got -> False
str: 'abc'       is a float? expected(False) got -> False
str: '123a'      is a float? expected(False) got -> False
str: '--123'     is a float? expected(False) got -> False
```

```
str: '123.45.67' is a float? expected(False) got -> False
str: '123.45.67' is a float? expected(False) got -> False
str: '123.45a'   is a float? expected(False) got -> False
str: '123a2'     is a float? expected(False) got -> False
str: '123e2.2'   is a float? expected(False) got -> False
str: '123    '   is a float? expected(False) got -> False
str: '  123  '   is a float? expected(False) got -> False

Matched 26 out of 26 test cases!

Converting to float!
123        : 123.0000000000
-123       : -123.0000000000
+123       : 123.0000000000
123.45     : 123.4500000000
.45        : 0.4500000000
-123.45    : -123.4500000000
+123.45    : 123.4500000000
1.23e-4    : 0.0001230000
-1.23E4    : -12,300.0000000000
1.23e+4    : 12,300.0000000000
1.23E-4    : 0.0001230000
1.23E4     : 12,300.0000000000
2E2        : 200.0000000000
2e2        : 200.0000000000
```

1.4 Representação do Grafo de saída correspondente:

```
from graphviz import Digraph

#(...)
tt {
   # (...)
}
#(...)

def graph():
   """ Generate a graph image that represents the automaton. """
   dot = Digraph("Automato Finito Deterministico")

   dot.format = "png"
   dot.attr("graph", dpi="300")

   for state in tt.keys():
      if state in F:
         shape = "doublecircle"
      else:
         shape = "circle"
      dot.node(state, state, shape=shape)

   for state, transitions in tt.items():
      for symbol, next_state in transitions.items():
         dot.edge(state, next_state, label=symbol)
```

```
   dot.view()

graph()
```

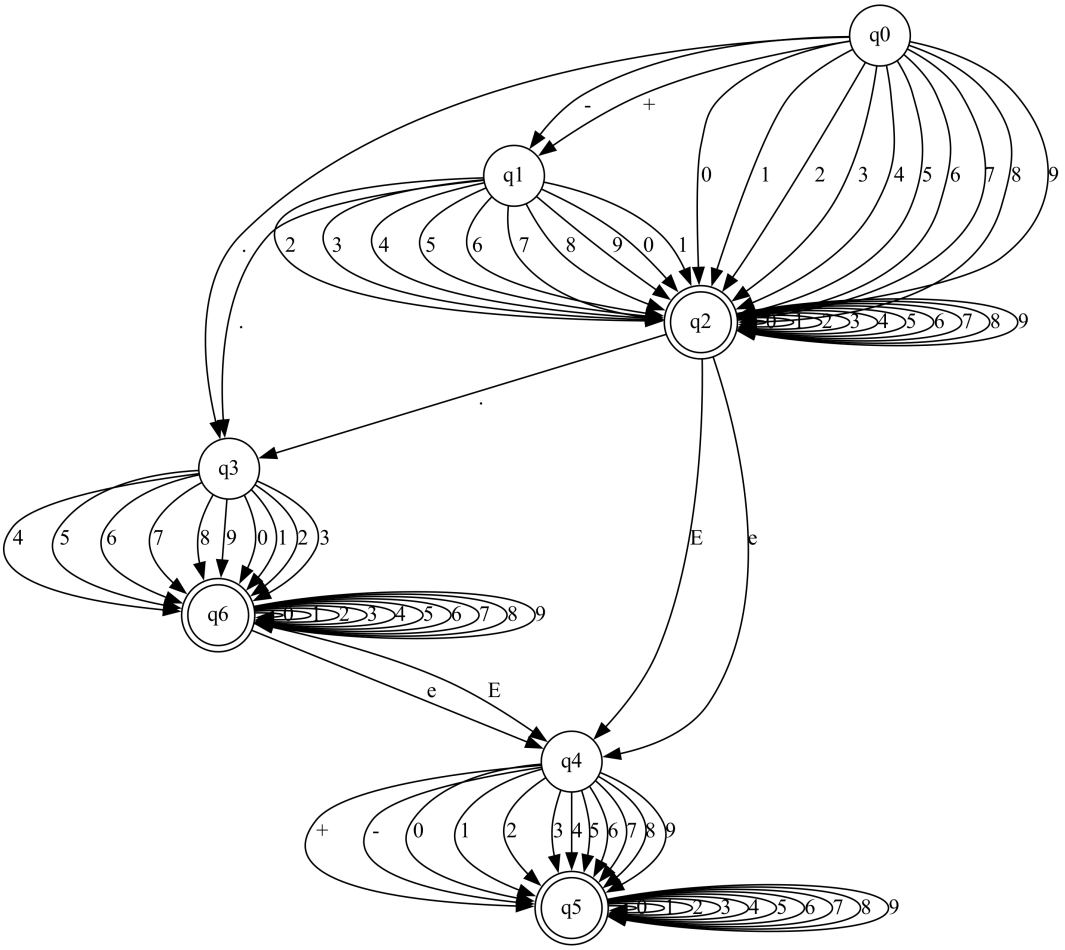

Figure 1.6: Automato Finito Determinista final