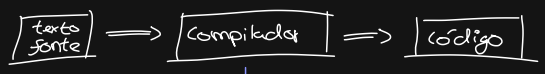
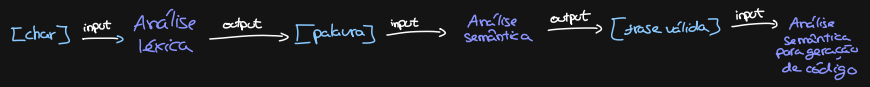
Linguagem

↳ vocabulário

↳ Sintaxe

↳ Semântica





Linguagens naturais → tipo 0

Linguagens com contexto → tipo 1

Linguagens independentes de contexto → tipo 2 → BNF

Linguagens regulares → tipo 3 → Expressões regulares

→ Autómatos Finitos Determinísticos (AFD) ← (AFND) → não computável

→ gramáticas regulares

Expressões regulares

↳ padrões

→ Exemplo: apanhar um inteiro

\+ ou - seguido de pelo menos um dígito

↳ expressão regular ⇒ ('+'|'-')? ('0'|'1'|'2'|...|'9')+

('0'|'1'|'2'|...|'9') = [0-9]

letras ⇒ [A-Za-z]+ ↳ para detectar palavras

Canais de saída

→ Stdin → teclado

→ Stdout → monitor

→ Stderr →

redirecionar:

```
$ python3 rec.py < texto.txt
$ cat texto.txt | python3 rec.py
```

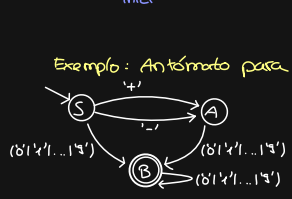
Consumir o Stdin:

```
import sys
for linha in sys.stdin:
    print(linha)
```

Autómato Finito Determinista (AFD)

AFD = (S, Σ, V, δ)

S: estados; Σ: estados; V: símbolos; δ: função de transição



estados	+	-	0	1	2	...	9
S_1	A_2	A_2	B_3	B_3	B_3		B_3
A_2	—	—	B_3	B_3	B_3		B_3
B_3	—	—	B_3	B_3	B_3		B_3

Exemplo: Expressões Regular

data → \d{4}-\d{2}-\d{2}

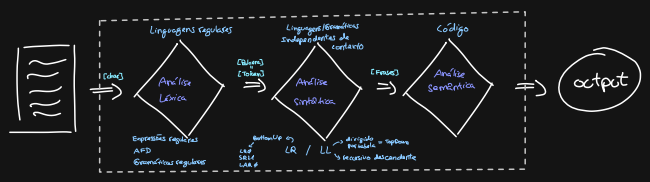
inteiro → [+\-]?\d{1,}

↳ para tirar o significado especial dos o '-' dentro do []

↳ sem limite indicado

16.02.2024

Compilador / Interpretador



Expressões Regulares

Inteiro = ('+' | '-')? [0-9]+
= [+\-]? [0-9]+
= [+\-]? \d+

Operadores

.	qq char exceto "\n"
*	O elemento que o procede pode aparecer zero ou mais vezes
+	O elemento que o procede deve aparecer um vez pelo menos ou mais
?	O elemento pode aparecer zero ou uma vez
[...]	Indica conjunto de caracteres permitidos
[^]	Indica conjunto de caracteres que não são permitidos
^	Âncora - Informa que a correspondência deve ocorrer no início da String ou linha
$	Âncora - Informa que a correspondência deve ocorrer no fim da String ou linha
{m,n}	m - nº mínimo de vezes que o padrão deve corresponder n - nº máximo de vezes que o padrão deve corresponder
{k}	k - nº exato de vezes que o padrão deve corresponder
( )	Grupo de captura
(?:)	Grupo não capturado
\1, \2, ..., \9	Referência a grupo previamente capturado

(operadores de contexto)

texto

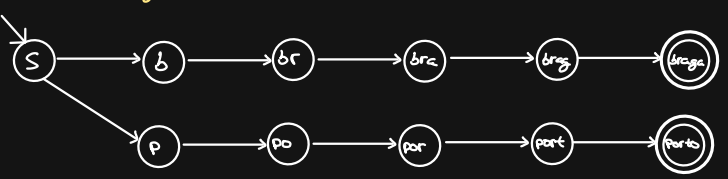
link = <[^>]*>

real = sinal? inteiro (\. decimal)? expoente?
↳ sinal = '+' | '-'
↳ inteiro = \d+
↳ decimal = \d+
↳ expoente = E sinal inteiro

+0.55E-7
sinal · inteiro · ponto · decimal · expoente

Automatro

Reconhecer uma sequência
e = r'braga | porto'



String binária

↳ ^ [01]* $

String binária que termina com 2 Øs

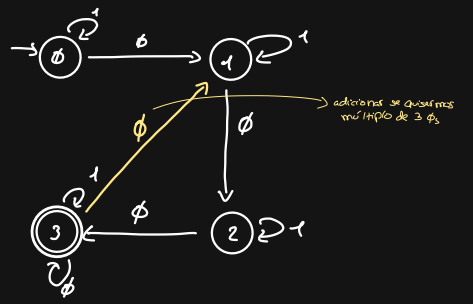
↳ 1 [01]+ ØØ $

String binária com pelo menos 1 Ø

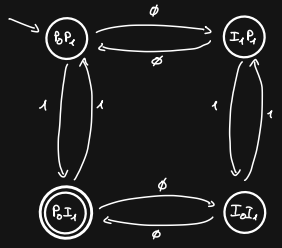
↳ [Ø1]* Ø [Ø1]*

Strings binárias com pelo menos 3 Øs

↳ [Ø1]* Ø [Ø1]* Ø [Ø1]* Ø [Ø1]*



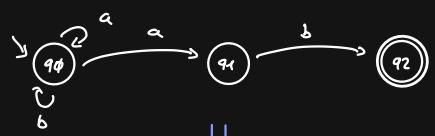
String binária com nº par de Øs e ímpar de 1s



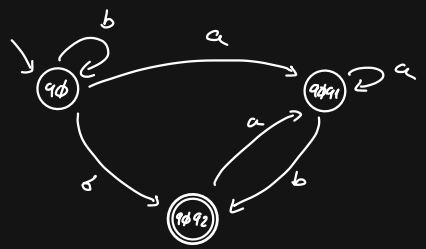
	Ø	1
> P_0P_1	I_0P_1	P_0I_1
I_0P_1	P_0P_1	I_0I_1
* P_0I_1	I_0I_1	P_0P_1
I_0I_1	P_0I_1	I_0P_1

Autómato Finito Não Determinista → Pode estar em vários estados simultaneamente durante o processo de uma entrada

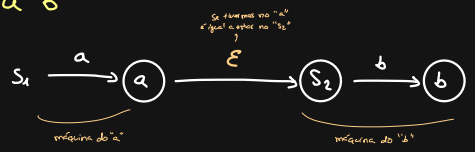
(a|b)*ab



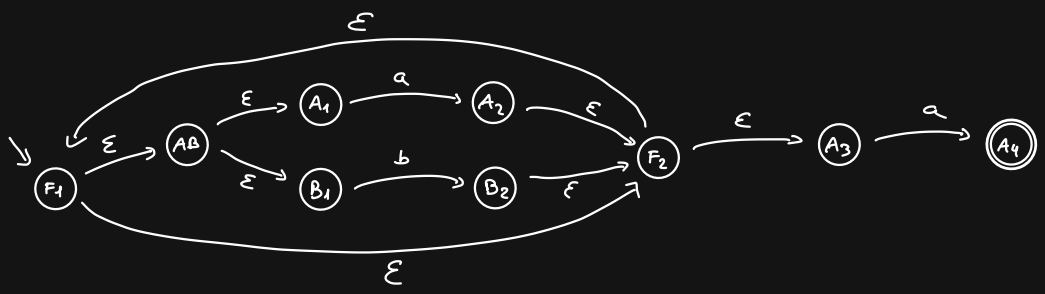
ε^*	a	b
q_0	q_0, q_1	q_0
q_0, q_1 (estando em q_0 ou q_1)	q_0, q_1	q_0, q_2
q_0, q_2	q_0, q_1	q_0



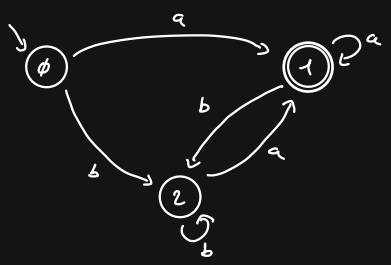
a b



(a|b)*a



ε^*	a	b
$F_1 = \phi$	$A_2, A_4 = 1$	$B_2 = 2$
$A_2, A_4 = 1$	$A_2, A_4 = 1$	$B_2 = 2$
$B_2 = 2$	$A_2, A_4 = 1$	$B_2 = 2$



23-02-2024

Greedy and Non Greedy Quantifiers

Greedy: Tentam reconher com o maior padrão possível → operadores: +, -, ...

Non Greedy: Se adicionarmos '?', ele passa a tentar reconher o menor padrão possível → operadores: +?, -?, ...

Grupos de captura

r'(\+|-)?(\d+)

→ group(0): ('-', '123')

→ group(1): -

→ group(2): 123

```
import re
import sys

url = r"\s*href=['\"]([^']*?)\1"
er11 = re.compile(url)

urls = []
for linha in sys.stdin:
    captura = er11.search(linha)
    if captura != None:
        urls.append(captura.group(1))

urls.sort()
print("\n".join(urls))
```

\1 para este grupo ser igual ao 1º

Grupos nomeados

↳ (?P<names...)

→ inteiro: (?P<sinal>[+\-])(?P<iss>\d+)

- groups() → +56 → ('+', '56')
- group('sinal') → +56 → +
- group('iss') → +56 → 56
- groupdict() → +56 → {'sinal': '+', 'iss': '78'}

Automato Finito Determinístico

```
class FiniteStateAutomaton:
    def __init__(self, states, alphabet, transitions, initial_state, accepting_states):
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions
        self.current_state = initial_state
        self.initial_state = initial_state
        self.accepting_states = accepting_states

    def reset(self):
        self.current_state = self.initial_state

    def process_input(self, inseq):
        error = False
        i = 0
        while (i < len(inseq)) and not error:
            if inseq[i] not in self.alphabet:
                error = True
                error_message = f"Erro: símbolo '{inseq[i]}' não pertence ao alfabeto."
            elif self.current_state not in self.transitions or inseq[i] not in self.transitions[self.current_state]:
                error = True
                error_message = f"Erro: não há transição definida para o estado '{self.current_state}' com o símbolo '{inseq[i]}'."

            else:
                self.current_state = self.transitions[self.current_state][inseq[i]]
            i = i + 1
        if error:
            return (False, error_message)
        else:
            if self.current_state in self.accepting_states:
                return (True, "Token válido.")
            else:
                return (False, "Erro semântico: o autómato não atingiu um estado final!")

# Example usage:

# Define states, alphabet, transitions, initial state, and accepting states
states = {'A', 'B', 'C'}
alphabet = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '-'}
transitions = {'A': {'0': 'C', '1': 'C', '2': 'C', '3': 'C', '4': 'C', '5': 'C', '6': 'C', '7': 'C', '8': 'C', '9': 'C', '+': 'B', '-': 'B'},
               'B': {'0': 'C', '1': 'C', '2': 'C', '3': 'C', '4': 'C', '5': 'C', '6': 'C', '7': 'C', '8': 'C', '9': 'C'},
               'C': {'0': 'C', '1': 'C', '2': 'C', '3': 'C', '4': 'C', '5': 'C', '6': 'C', '7': 'C', '8': 'C', '9': 'C'}}
initial_state = 'A'
accepting_states = {'C'}

# Create the Finite State Automaton
fsa = FiniteStateAutomaton(states, alphabet, transitions, initial_state, accepting_states)

# Process input sequences
input_sequence1 = '+123'
result1 = fsa.process_input(input_sequence1)
print(f"Result for '{input_sequence1}': {result1[0]}: {result1[1]}")

fsa.reset()
input_sequence2 = '-110'
result2 = fsa.process_input(input_sequence2)
print(f"Result for '{input_sequence2}': {result2[0]}: {result2[1]}")
```

Análise Léxica

=> conjunto T = {...} → é o conjunto dos símbolos terminais / tokens

- → Sinais: São constituidos por um caracter
- → Palavras reservadas: Strings constantes
- → Terminais variáveis: Identificadores, inteiros, etc.

15.03.2024

GIC → Gramáticas Independentes de Contexto

T: Símbolos Terminais

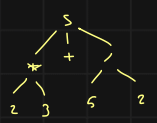
N: Não-Terminais

S: Ponto de Partida

P: Produções / Regras

Para especificar a gramática devemos ir das operações com menos prioridade para as que tem mais.

2 * 3 + 5 / 2



Exp → Termo `+' Exp
| Termo `-' Exp
| Termo

ambígua

Termo → Fator '*' Termo
| Fator '/' Termo
| Fator

ambígua

Fator → `(' Exp ')' | num | id

[]

[1]

[1,2,3]

Lista $\xrightarrow{P_1}$ '[' cont

Cont $\xrightarrow{P_2}$ num Cont2 ']'

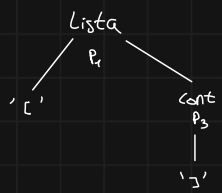
P_3 | ']'

Cont2 $\xrightarrow{P_4}$ ',' num Cont2

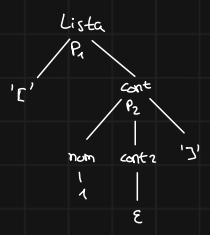
P_5 | ε

Árvores de derivação ou de sintaxe abstrata (ASA)

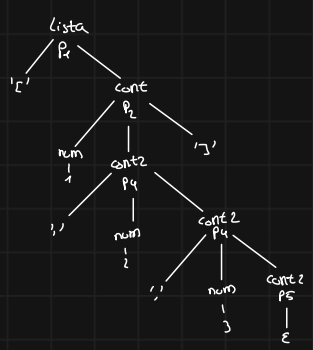
[ ]



[1]



[1,2,3]



Top Down

LL(1): dirigido por tabela

Recursivo Descendente (LL(1))

→ o 1º L: lemos o input da esq. para a dir.

→ o 2º L: andamos da esq. para a dir. na gramática

Bottom Up

LR(0)

SLR(1)

LALR(0)

Gramáticas LL(1)

↳ quer dizer que só usamos um valor a seguir para tomar a decisão

Lista → '[' ']'
| '[' cont ']'

↳ prefixo comum → por em evidência

Lista → '[' Lista2
Lista2 → ']'
| cont ']'

cont → num cont2

cont2 → ',' num cont2
| ε

→ Tem que cumprir a condição LL(1)

$\forall p_1, p_2 \in P \; L_A(p_1) \cap L_A(p_2) = \emptyset$

Lista $\xrightarrow{P_1}$ '[' cont $L_A = \{'['\}$ → o 1º símbolo terminal

cont $\xrightarrow{P_2}$ num cont2 ']' $L_A = \{num\}$

P_3 | ']' $L_A = \{']'\}$

cont2 $\xrightarrow{P_4}$ ',' num cont2 $L_A = \{','\}$

P_5 | ε $L_A = \{']'\}$

↳ $L_A(cont_2 \to \varepsilon)$ = Follow(cont2) = {']'}

↳ olhar para o lado direito, onde aparece o cont2 e ver qual o símbolo terminal que vem a seguir

outro exemplo

Z → S $

S $\xrightarrow{P_1}$ '(' S ')' S {'('}

P_2 | ε {')', '$'}

($ fim do ficheiro; como não tem acrescenta)

outro exemplo

S → Sa
| b

recursividade à esquerda → criar outro não terminal

S → S'a {b}
| b {b}

$L_A(S \to Sa) = First_N(S) = First(Sa) \cup First(b)$

(First(Sa): ↓ recursivo; First(b): ↓ símbolo terminal)

$= \{b\}$