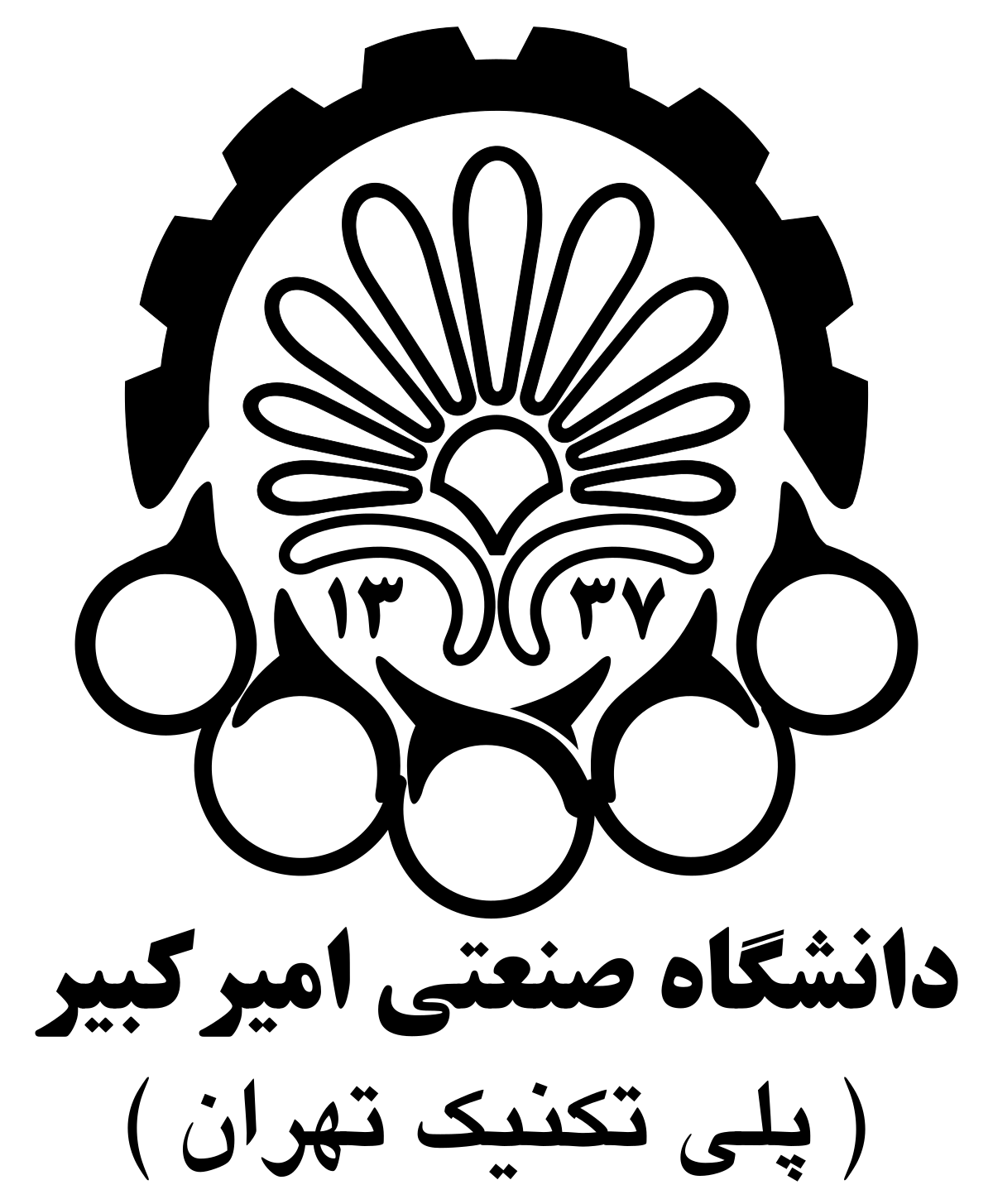
**بسم تعالی**

****

کیان پورآذر

۴۰۱۳۱۴۰۳

استاد:

خانم دکتر عامری

**فهرست مطالب**

سوال ۱ .........................................................................................۳

سوال ۲ .........................................................................................۷

سوال ۳ .........................................................................................۱۱

سوال ۱: سیستمی را طراحی و پیاده سازی نمایید که عبارت منظم را به پذیرنده متناهی غیر قطعی NFAتبدیل نماید.

**معرفی کلاس‌ها و توابع:**

**۱ .کلاس “state” :**

- این کلاس برای نشان دادن هر حالت در NFA استفاده می‌شود.

- دارای ویژگی‌های زیر است:

:”name” - نام یا شماره حالت.

: “transitions” - یک دیکشنری که کلیدهای آن سمبل‌های ورودی و مقادیر آن لیستی از حالت‌هایی است که با آن سمبل به آنها انتقال می‌یابیم.

:”is\_final” - یک بولین برای نشان دادن اینکه آیا این حالت پایانی است یا خیر.

- متد: “add\_transition(symbol, state)” یک انتقال جدید به حالت اضافه می‌کند. اگر سمبل قبلاً وجود داشته باشد، حالت جدید به لیست حالت‌های موجود اضافه می‌شود، در غیر این صورت یک لیست جدید ایجاد می‌شود.

class State:

def \_\_init\_\_(self, name):

self.name = name

self.transitions = {}

self.is\_final = False

def add\_transition(self, symbol, state):

if symbol in self.transitions:

self.transitions[symbol].append(state)

else:

self.transitions[symbol] = [state]

**2 . کلاس "NFA"**

- این کلاس برای نمایش یک NFA استفاده می‌شود که دارای ویژگی‌های زیر است:

:"start\_state" - حالت شروع .NFA

- "final\_state": حالت پایانی NFA.

class NFA:

def \_\_init\_\_(self, start\_state, final\_state):

self.start\_state = start\_state

self.final\_state = final\_state

**3 .تابع :“regex\_to\_nfa”**

- این تابع عبارت منظم را به عنوان ورودی می‌گیرد و یک NFA معادل با آن تولید می‌کند.

- از یک پشته برای نگهداری NFA‌های جزئی استفاده می‌کند.

- هر کاراکتر از عبارت منظم به صورت جداگانه پردازش می‌شود:

اگر کاراکتر یک عملگر “\*” باشد، NFA بالای پشته را تغییر می‌دهد تا حالت(“\*”) “Closure” را اعمال کند.

اگر کاراکتر یک عملگر “|” باشد، دو NFA از بالای پشته خارج می‌کند و آنها را با هم ترکیب می‌کند.

اگر کاراکتر یک عملگر “.” باشد، دو NFA از بالای پشته خارج می‌کند و آنها را پشت سر هم قرار می‌دهد.

در غیر این صورت، یک NFA جدید برای کاراکتر ایجاد می‌کند.

در انتها، NFA نهایی از پشته خارج شده و بازگردانده می‌شود.

def regex\_to\_nfa(regex):

stack = []

state\_count = 0

def new\_state():

nonlocal state\_count

state\_count += 1

return State(f"q{state\_count}")

alphabet = set()

for char in regex:

if char.isalnum():

alphabet.add(char)

if char == '\*':

nfa = stack.pop()

start\_state = new\_state()

final\_state = new\_state()

start\_state.add\_transition('', nfa.start\_state)

nfa.final\_state.add\_transition('', final\_state)

nfa.final\_state.add\_transition('', nfa.start\_state)

start\_state.add\_transition('', final\_state)

stack.append(NFA(start\_state, final\_state))

elif char == '|':

nfa2 = stack.pop()

nfa1 = stack.pop()

start\_state = new\_state()

final\_state = new\_state()

start\_state.add\_transition('', nfa1.start\_state)

start\_state.add\_transition('', nfa2.start\_state)

nfa1.final\_state.add\_transition('', final\_state)

nfa2.final\_state.add\_transition('', final\_state)

stack.append(NFA(start\_state, final\_state))

elif char == '.':

nfa2 = stack.pop()

nfa1 = stack.pop()

nfa1.final\_state.add\_transition('', nfa2.start\_state)

stack.append(NFA(nfa1.start\_state, nfa2.final\_state))

else:

start\_state = new\_state()

final\_state = new\_state()

start\_state.add\_transition(char, final\_state)

stack.append(NFA(start\_state, final\_state))

final\_nfa = stack.pop()

final\_nfa.final\_state.is\_final = True

return alphabet, final\_nfa

**.4 تابع :“print\_nfa”**

- این تابع یک NFA را دریافت کرده و انتقال‌های آن را چاپ می‌کند.

- از یک لیست به عنوان پشته برای پیمایش گراف NFA استفاده می‌کند.

- هر حالت که پردازش می‌شود، در مجموعه “visited” ثبت می‌شود تا از بازدید مکرر جلوگیری شود.

- برای هر انتقال از یک حالت به حالت دیگر، خطی در خروجی چاپ می‌شود که نشان دهنده حالت مبدأ، سمبل انتقال و حالت مقصد است.

def print\_nfa(alphabet, nfa):

states = set()

transitions = []

final\_states = set()

def traverse(state):

if state.name in states:

return

states.add(state.name)

if state.is\_final:

final\_states.add(state.name)

for symbol, next\_states in state.transitions.items():

for next\_state in next\_states:

transitions.append((state.name, symbol, next\_state.name))

traverse(next\_state)

traverse(nfa.start\_state)

print(" ".join(alphabet))

print(" ".join(sorted(states, key=lambda x: int(x[1:]))))

print(nfa.start\_state.name)

print(" ".join(sorted(final\_states, key=lambda x: int(x[1:]))))

for (src, symbol, dst) in transitions:

print(f"{src} {symbol if symbol else 'λ'} {dst}")

**مثال:**

**ورودی:** “ab\*a”

regex = input() #input: "ab\*a"

alphabet, nfa = regex\_to\_nfa(regex)

print\_nfa(alphabet, nfa)

**خروجی:**

a b

q7 q8

q7

q8

q7 a q8

سوال 2: سیستمی را طراحی و پیاده سازی نمایید که گرامر مستقل از متن را به عنوان ورودی بگیرد و برای آن ماشین پشته ای غیرقطعی معادلش را به عنوان خروجی برگرداند. توجه کنید که گرامر مستقل از متن لزوما به فرم های نرمال نیست.

برای طراحی و پیاده‌سازی سیستمی که گرامر مستقل از متن ((Context-Free Grammer یا CFG را به عنوان ورودی میگیرد و ماشین پشته‌ای غیرقطعی ((Nondeterministic Pushdown Automaton یا NPDA معادل آن را به عنوان خروجی برمیگرداند، طبق مراحل زیر:

**1. ورودی گرفتن گرامر مستقل از متن**

- گرامر را از کاربر دریافت کرده و آن را به صورت فرمت استاندارد ذخیره میکند. گرامر شامل مجموعه‌ای از قواعد تولید است که هر کدام از یک نماد غیرپایانی به یک رشته از نمادهای پایانی و غیرپایانی تولید می‌شوند.

**2. تعریف ماشین پشته‌ای غیرقطعی**

- یک ماشین پشته‌ای غیرقطعی را با استفاده از عناصر زیر تعریف میکنیم:

- مجموعه‌ای از حالات

- الفبای ورودی

- الفبای پشته

- حالت شروع

- نماد شروع پشته

- مجموعه‌ای از حالات پایانی

- مجموعه‌ای از انتقالات

**3. تبدیل گرامر به ماشین پشته‌ای**

- الگوریتم تبدیل هر قاعده تولید گرامر به انتقالات در ماشین پشته‌ای را پیاده‌سازی میکنیم. این تبدیل باید به گونه‌ای باشد که هر رشته‌ای که توسط گرامر تولید می‌شود، توسط ماشین پشته‌ای نیز پذیرفته شود.

**4. پیاده‌سازی و خروجی ماشین پشته‌ای**

- ماشین پشته‌ای غیرقطعی را پیاده‌سازی میکنیم و نتایج آن را به کاربر نمایش میدهیم.

**توضیحات کد:**

1. کلاس “CFGtoNPDA”

- این کلاس گرامر مستقل از متن (CFG) را به عنوان ورودی گرفته و ماشین پشته‌ای غیرقطعی (NPDA) معادل آن را تولید می‌کند.

- کلاس با دریافت نمادهای پایانی، نمادهای غیرپایانی، نماد شروع، و قواعد تولید، مقادیر اولیه را تنظیم می‌کند.

class CFGtoNPDA:

def \_\_init\_\_(self, terminals, non\_terminals, start\_symbol, productions):

self.terminals = terminals

self.non\_terminals = non\_terminals

self.start\_symbol = start\_symbol

self.productions = productions

self.states = set()

self.transitions = []

self.start\_state = '0'

self.accept\_state = '-1'

self.stack\_start\_symbol = '$'

2. متد “create\_npda”

- این متد برای ایجاد NPDA بر اساس گرامر ورودی استفاده می‌شود.

- ابتدا حالات و انتقالات اولیه را تعریف می‌کند. حالات شامل q0، q\_push، q\_pop و q\_accept هستند.

- انتقال اولیه از حالت q0 به q\_push اضافه می‌شود که نماد شروع گرامر و نماد شروع پشته را به پشته اضافه می‌کند.

def create\_npda(self):

self.states = {'0', '1', '2', '-1'}

self.transitions.append((0, '', None, 1, f'PUSH({self.start\_symbol}{self.stack\_start\_symbol})'))

for lhs, rhs\_list in self.productions.items():

for rhs in rhs\_list:

self.transitions.append((1, '', lhs, 1, f'PUSH({rhs})'))

for terminal in self.terminals:

self.transitions.append((1, terminal, terminal, 2, 'POP'))

self.transitions.append((1, 'ε', self.stack\_start\_symbol, 2, 'POP'))

self.transitions.append((2, 'ε', None, -1, 'NONE'))

3. اضافه کردن قواعد تولید گرامر به انتقالات

- برای هر قاعده تولید گرامر (LHS -> RHS)، یک انتقال به NPDA اضافه می‌شود.

- این انتقال از حالت q\_push به حالت q\_push می‌رود و LHS را از پشته برداشته و RHS را به پشته اضافه می‌کند.

4. اضافه کردن انتقالات برای نمادهای پایانی

- برای هر نماد پایانی، یک انتقال اضافه می‌شود که نماد پایانی ورودی را با نماد پایانی پشته تطبیق داده و از پشته برمی‌دارد.

5. اضافه کردن انتقال برای پذیرش

- انتقالی از حالت q\_pop به حالت q\_accept اضافه می‌شود که زمانی اتفاق می‌افتد که نماد شروع پشته ($) از پشته برداشته شود.

6. متد “display\_npda”

- این متد برای نمایش حالات و انتقالات NPDA استفاده می‌شود.

- شامل نمایش حالات، حالت شروع، حالت پذیرش و انتقالات است.

def display\_npda(self):

for transition in self.transitions:

print(", ".join(map(str, transition)))

۷.متد “parse\_input”

- این تابع ورودی کاربر را تجزیه و تحلیل کرده و نمادهای پایانی، نمادهای غیرپایانی، نماد شروع و قواعد تولید را استخراج می‌کند.

def parse\_input(lines):

terminals = set()

non\_terminals = set()

production\_rules = {}

start\_symbol = None

for line in lines:

line = line.strip()

if not line:

continue

lhs, rhs = line.split('[')

lhs = lhs.strip()

rhs = rhs.strip(']').split(',')

if start\_symbol is None:

start\_symbol = lhs

if lhs not in production\_rules:

production\_rules[lhs] = []

production\_rules[lhs].extend(rhs)

non\_terminals.add(lhs)

for symbol in rhs:

if symbol.islower():

terminals.add(symbol)

elif symbol.isupper():

non\_terminals.add(symbol)

elif symbol == 'ε':

production\_rules[lhs].append('')

return terminals, non\_terminals, start\_symbol, production\_rules

**مثال:**

**ورودی:**

print("Enter the grammar rules (empty line to finish):") #input: S [a,A,b]

input\_lines = [] #A [a,S,b]

while True: #A [ε]

line = input()

if line.strip() == '':

break

input\_lines.append(line)

terminals, non\_terminals, start\_symbol, productions = parse\_input(input\_lines)

cfg\_to\_npda = CFGtoNPDA(terminals, non\_terminals, start\_symbol, productions)

cfg\_to\_npda.create\_npda()

cfg\_to\_npda.display\_npda()

**خروجی:**

0, , None, 1, PUSH(S$)

1, , S, 1, PUSH(a)

1, , S, 1, PUSH(A)

1, , S, 1, PUSH(b)

1, , A, 1, PUSH(a)

1, , A, 1, PUSH(S)

1, , A, 1, PUSH(b)

1, , A, 1, PUSH(ε)

1, ε, ε, 2, POP

1, b, b, 2, POP

1, a, a, 2, POP

1, ε, $, 2, POP

2, ε, None, -1, NONE

سوال ۳: ماشین تورینگی طراحی و پیاده سازی کنید که دو رشته را به عنوان ورودی دریافت نموده و بزرگترین زیر رشته مشترک بین آن ها را پیدا کند.

برای پیاده‌سازی ماشین تورینگ که بزرگترین زیررشته مشترک بین دو رشته ورودی را پیدا کند، می‌توان از یک الگوریتم مشابه الگوریتم برنامه‌ریزی پویا برای یافتن بزرگترین زیررشته مشترک (Longest Common Subsequence) استفاده کرد.

**مراحل طراحی ماشین تورینگ:**

1. تعریف نوارها و الفبای ماشین تورینگ

- نوار اول: رشته اول (s1).

- نوار دوم: رشته دوم (s2).

- نوار سوم: برای ذخیره‌سازی نتایج میانی و نهایی.

2. الفبای ماشین

- شامل تمام کاراکترهای رشته‌های ورودی.

- کاراکترهای ویژه برای نشانه‌گذاری (مثل # برای جدا کردن بخش‌ها)

3. حالات ماشین

- حالت اولیه (q0): دریافت و بررسی کاراکترهای ورودی.

- حالت‌های مقایسه: مقایسه کاراکترهای رشته‌ها.

- حالت‌های حرکت: حرکت به سمت راست یا چپ روی نوارها.

- حالت‌های ذخیره‌سازی: ذخیره نتایج میانی.

- حالت نهایی (q\_accept): ماشین نتیجه را روی نوار خروجی نوشته و متوقف می‌شود.

**الگوریتم پیاده‌سازی:**

1. مرحله آماده‌سازی

- ماشین رشته‌های ورودی را روی نوارهای اول و دوم قرار می‌دهد و سرهای نوار را به ابتدای رشته‌ها می‌برد.

2. مقایسه کاراکترها

- ماشین تورینگ با استفاده از یک حلقه تودرتو، کاراکترهای هر دو رشته را یکی یکی مقایسه می‌کند.

- اگر کاراکترهای مشابه پیدا شوند، موقعیت آن‌ها ذخیره می‌شود.

3. ذخیره‌سازی نتایج میانی

- اگر یک کاراکتر مشابه پیدا شد، طول زیررشته مشترک محاسبه و در نوار سوم ذخیره می‌شود.

4. حرکت و بررسی سایر زیررشته‌ها

- ماشین به حرکت در نوارها ادامه می‌دهد و سایر زیررشته‌ها را بررسی می‌کند.

5. پیدا کردن بزرگترین زیررشته مشترک

- بعد از مقایسه تمامی کاراکترها، ماشین تورینگ طول بزرگترین زیررشته مشترک را محاسبه و نتیجه را روی نوار خروجی می‌نویسد.

**توضیحات کد:**

1. تابع “longest\_common\_subsequence”

- این تابع دو رشته “s1” و “s2” را به عنوان ورودی می‌گیرد و بزرگترین زیررشته مشترک را بازمی‌گرداند.

- ابتدا یک جدول “lcs\_lengths” ایجاد می‌شود که به ازای هر زوج (i, j) ، مقدار lcs\_lengths[i][j] بیانگر طول بزرگترین زیررشته مشترک بین s1[0...i-1] و s2[0...j-1] است.

2. ساخت جدول “lcs\_lengths”

- با استفاده از یک حلقه دوتایی، مقادیر جدول “lcs\_lengths” را به صورت پایین به بالا و از چپ به راست پر می‌کنیم.

- اگر کاراکترهای s1[i-1] و s2[j-1] برابر باشند، lcs\_lengths[i][j] برابر با lcs\_lengths[i-1][j-1] + 1 خواهد بود؛ در غیر این صورت، برابر با max(lcs\_lengths[i-1][j], lcs\_lengths[i][j-1]) قرار می‌گیرد.

3. ساخت زیررشته مشترک

- بعد از پر کردن جدول “lcs\_lengths”، طول بزرگترین زیررشته مشترک در `lcs\_lengths[m][n]` قرار دارد.

- برای بازیابی خود زیررشته، از انتهای جدول به ابتدا حرکت کرده و با استفاده از اعداد موجود در آن، عناصر زیررشته مشترک را برمی‌گردانیم.

def longest\_common\_subsequence(s1, s2):

m = len(s1)

n = len(s2)

lcs\_lengths = [[0] \* (n + 1) for \_ in range(m + 1)]

for i in range(1, m + 1):

for j in range(1, n + 1):

if s1[i - 1] == s2[j - 1]:

lcs\_lengths[i][j] = lcs\_lengths[i - 1][j - 1] + 1

else:

lcs\_lengths[i][j] = max(lcs\_lengths[i - 1][j], lcs\_lengths[i][j - 1])

length\_lcs = lcs\_lengths[m][n]

lcs = []

i, j = m, n

while i > 0 and j > 0:

if s1[i - 1] == s2[j - 1]:

lcs.append(s1[i - 1])

i -= 1

j -= 1

elif lcs\_lengths[i - 1][j] >= lcs\_lengths[i][j - 1]:

i -= 1

else:

j -= 1

lcs.reverse()

lcs\_str = ''.join(lcs)

return lcs\_str

4. مثال استفاده:

- در این مثال ، دو رشته "abcdefg" و "abghxfwg" به عنوان ورودی داده شده‌اند و زیررشته مشترک بزرگترین آنها "abfg" است که در خروجی نمایش داده می‌شود.

**مثال:**

**ورودی:**

s1 = input("first:") #input: "abcdefg"

s2 = input("second:") #input: "abghxfwg"

result = longest\_common\_subsequence(s1, s2)

print(f"The longest common subsequence is: {result}")

**خروجی:**

The longest common subsequence is: "abfg"