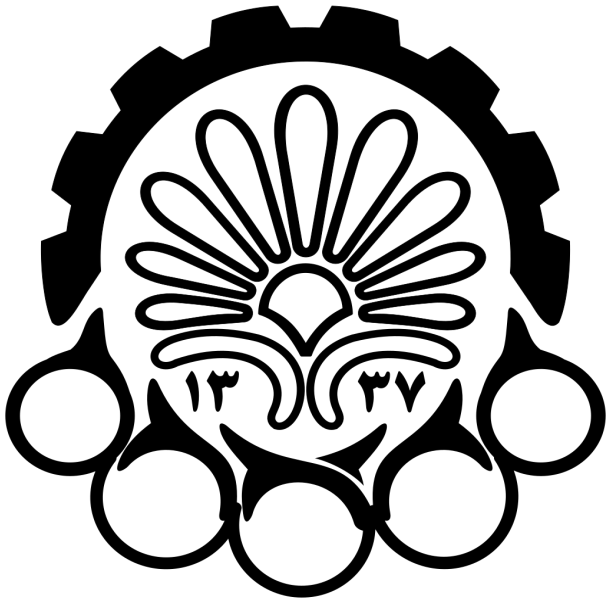
بسم تعالی



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

کیان پورآذر

۴۰۱۳۱۴۰۳

استاد:

خانم دکتر عامری

فهرست مطالب

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]
```

۲ . 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
```

۳. 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 NFA های 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
```

۴. 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 نشان‌دهنده 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 ۲: 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:

۱ .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.

۲ .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

۳ . 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.

۴ .پیاده‌سازی و خروجی ماشین پشته‌ای

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

توضیحات کد:

۱ .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 = '$'
```

۲ .متد"create_npda"

- این متد برای ایجاد NPDA بر اساس گرامر ورودی استفاده می‌شود.
- ابتدا حالات و انتقالات اولیه را تعریف می‌کند. حالات شامل q۰ ، q_push، q_pop و q_accept هستند.
- انتقال اولیه از حالت q۰ 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'))
```

۳ .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.

۴ .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.

۵ .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.

۶ .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، می‌تواند از یک الگوریتم مشابه الگوریتم برنامه‌ریزی پویا برای یافتن بزرگترین زیررشته مشترک (Longest Common Subsequence) استفاده کرد.

مراحل طراحی ماشین تورینگ:

۱ .تعریف نوارها و الفبای ماشین تورینگ

- نوار اول: رشته اول(S۱)
- نوار دوم: رشته دوم.(S۲)
- نوار سوم: برای ذخیره‌سازی نتایج میانی و نهایی.

۲ .الفبای ماشین

- شامل تمام کاراکترهای رشته‌های ورودی.
- کاراکترهای ویژه برای نشانه‌گذاری (مثل # برای جدا کردن بخش‌ها)

۳ .حالات ماشین

- حالت اولیه :(q۰) 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:

۱ .مرحله آماده‌سازی

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

۲ .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.

۳ .ذخیره‌سازی نتایج میانی

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

۴ .حرکت و بررسی سایر زیررشته‌ها

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

۵ .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:

۱. تابع "longest_common_subsequence"

- این تابع دو رشته "s۱" و "s۲" 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 s۱[۰...i-۱] و s۲[۰...j-۱] است.

۲. 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 s۱[i-۱] و s۲[j-۱] برابر باشند، lcs_lengths[i][j]برابر با ۱ + lcs_lengths[i-۱][j-۱] خواهد بود؛ در غیر این صورت، برابر با max(lcs_lengths[i-۱][j], lcs_lengths[i][j-۱]) قرار می‌گیرد.

۳. 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
```

۴ .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"
```