chmielewski_lab2

March 26, 2025

1 Zadanie 1

Ekstrakcja informacji z publikacji naukowych

[ ]:
```
import re
from typing import Optional


def parse_publication(reference: str) -> Optional[dict]:
    """
    Parse academic publication reference and extract structured information.

    Expected reference format:
    Lastname, I., Lastname2, I2. (Year). Title. Journal, Volume(Issue),␣
 ↪StartPage-EndPage.

    Example:
    Kowalski, J., Nowak, A. (2023). Analiza algorytmów tekstowych. Journal of␣
 ↪Computer Science, 45(2), 123-145.

    Args:
        reference (str): Publication reference string

    Returns:
        Optional[dict]: A dictionary containing parsed publication data or None␣
 ↪if the reference doesn't match expected format
    """
    # TODO: Implement regex patterns to match different parts of the reference
    # You need to create patterns for:
    # 1. Authors and year pattern
    # 2. Title and journal pattern
    # 3. Volume, issue, and pages pattern
    authors_year_pattern = r"^(.*?)\s*\((\d{4})\)\."
    title_journal_pattern = r"\s*(.*?)\.\s*(.*?),"
    volume_issue_pages_pattern = r"\s*(\d+)(?:\((\d+)\))?,\s*(\d+)-(\d+)\.$"

    # TODO: Combine the patterns
```

```python
    # full_pattern = authors_year_pattern + title_journal_pattern + volume_issue_pages_pattern
    full_pattern = authors_year_pattern + title_journal_pattern + volume_issue_pages_pattern

    # TODO: Use re.match to try to match the full pattern against the reference
    # If there's no match, return None
    match = re.match(full_pattern, reference.strip())
    if not match: return None

    # TODO: Extract information using regex
    # Each author should be parsed into a dictionary with 'last_name' and 'initial' keys

    # TODO: Create a pattern to match individual authors
    author_pattern = r"^\s*([^.]+?),\s*([A-Z])\.?$"

    # TODO: Use re.finditer to find all authors and add them to authors_list
    authors_list = []
    authors_str = match.group(1).strip()
    for author in re.split(r"(\.,)", authors_str):
        author_match = re.match(author_pattern, author)
        if author_match:
            authors_list.append({'last_name' : author_match.group(1).strip(), 'initial' : author_match.group(2).strip()})

    # TODO: Create and return the final result dictionary with all the parsed information
    # It should include authors, year, title, journal, volume, issue, and pages

    result = {
        'authors' : authors_list,
        'year' : int(match.group(2)),
        'title' : match.group(3).strip(),
        'journal' : match.group(4).strip(),
        'volume' : int(match.group(5)),
        'issue' : int(match.group(6)) if match.group(6) else None,
        'pages' : {
            'start' : int(match.group(7)),
            'end' : int(match.group(8))
        }
    }

    return result
```

2 Zadanie 2

Analiza linków w kodzie HTML

[ ]:
```python
import re

def extract_links(html: str) -> list[dict[str, str]]:
    """
    Extract all links from the given HTML string.

    Args:
        html (str): HTML content to analyze

    Returns:
        list[dict]: A list of dictionaries where each dictionary contains:
            - 'url': the href attribute value
            - 'title': the title attribute value (or None if not present)
            - 'text': the text between <a> and </a> tags
    """

    # TODO: Implement a regular expression pattern to extract links from HTML.
    # The pattern should capture three groups:
    # 1. The URL (href attribute value)
    # 2. The title attribute (which might not exist)
    # 3. The link text (content between <a> and </a> tags)

    pattern = r"<a\s*href=\"([^\"]+)\"(?:\s+title=\"([^\"]*)\")?>(.*?)</a>"

    links = []

    # TODO: Use re.finditer to find all matches of the pattern in the HTML
    # For each match, extract the necessary information and create a dictionary
    # Then append that dictionary to the 'links' list

    matches = re.finditer(pattern, html)
    for match in matches:
        links.append({
            'url' : match.group(1),
            'title' : match.group(2),
            'text' : match.group(3)
        })

    return links
```

3 Zadanie 3

Analiza pliku tekstowego

[ ]:

```python
import re
from collections import Counter


def analyze_text_file(filename: str) -> dict:
    try:
        with open(filename, "r", encoding="utf-8") as file:
            content = file.read()
    except Exception as e:
        return {"error": f"Could not read file: {str(e)}"}

    # Common English stop words to filter out from frequency analysis
    stop_words = {
        "the",
        "a",
        "an",
        "and",
        "or",
        "but",
        "in",
        "on",
        "at",
        "to",
        "for",
        "with",
        "by",
        "about",
        "as",
        "into",
        "like",
        "through",
        "after",
        "over",
        "between",
        "out",
        "of",
        "is",
        "are",
        "was",
        "were",
        "be",
        "been",
        "being",
        "have",
        "has",
        "had",
        "do",
```

```
        "does",
        "did",
        "this",
        "that",
        "these",
        "those",
        "it",
        "its",
        "from",
        "there",
        "their",
    }

    # TODO: Implement word extraction using regex
    # Find all words in the content (lowercase for consistency)
    words = re.findall(r"\b[^\W\d_]+\b", content.lower())
    word_count = len(words)

    # TODO: Implement sentence splitting using regex
    # A sentence typically ends with ., !, or ? followed by a space
    # Be careful about abbreviations (e.g., "Dr.", "U.S.A.")
    sentence_pattern = r"[A-Z](?:.*?)(?<!Prof)(?<!\.)[.!?](?=\s+[A-Z]|\s*$)"
    sentences = re.findall(sentence_pattern, content, re.MULTILINE)
    sentence_count = len(sentences)

    # TODO: Implement email extraction using regex
    # Extract all valid email addresses from the content
    email_pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b"
    emails = re.findall(email_pattern, content)

    # TODO: Calculate word frequencies
    # Count occurrences of each word, excluding stop words and short words
    # Use the Counter class from collections
    words = re.findall(r'\b(?!' + '|'.join(re.escape(word) + r'\b' for word in stop_words) + r')[^\W\d_]{2,}\b', content.lower())
    frequent_words = dict(Counter(words).most_common(10))

    # TODO: Implement date extraction with multiple formats
    # Detect dates in various formats: YYYY-MM-DD, DD.MM.YYYY, MM/DD/YYYY, etc.
    # Create multiple regex patterns for different date formats
    date_patterns = [r"\d{4}-\d{2}-\d{2}", r'\d{1,2}\.\d{1,2}\.\d{4}', r'\d{1,2}/\d{1,2}/\d{4}', r'\b\d{2}-\d{2}-\d{4}\b', r'\b[\w]+\s{1}\d+,\s\d+\b']
    dates = []
    for pattern in date_patterns:
        dates.extend(re.findall(pattern, content))
```

```python
    # TODO: Analyze paragraphs
    # Split the content into paragraphs and count words in each
    # Paragraphs are typically separated by one or more blank lines
    paragraphs = re.split(r"\n\s*\n", content)
    words_pattern = r"\b[^\W\d_]+\b"
    paragraph_sizes = {}

    for i, paragraph in enumerate(paragraphs):
        words_in_paragraph = re.findall(words_pattern, paragraph)
        paragraph_sizes[i] = len(words_in_paragraph)

    return {
        "word_count": word_count,
        "sentence_count": sentence_count,
        "emails": emails,
        "frequent_words": frequent_words,
        "dates": dates,
        "paragraph_sizes": paragraph_sizes,
    }
```

4 Zadanie 4

Implementacja uproszczonego parsera regexpów

[ ]:
```python
from abc import ABC, abstractmethod
from collections import deque
from typing import Optional


class RegEx(ABC):
    @abstractmethod
    def nullable(self):
        pass

    @abstractmethod
    def derivative(self, symbol):
        pass

    def __eq__(self, other):
        if not isinstance(other, RegEx):
            return False
        return str(self) == str(other)

    def __hash__(self):
        return hash(str(self))
```

```python
class Empty(RegEx):
    def nullable(self):
        return False

    def derivative(self, symbol):
        return Empty()

    def __str__(self):
        return " "


class Epsilon(RegEx):
    def nullable(self):
        return True

    def derivative(self, symbol):
        return Empty()

    def __str__(self):
        return " "


class Symbol(RegEx):
    def __init__(self, symbol):
        self.symbol = symbol

    def nullable(self):
        return False

    def derivative(self, symbol):
        if self.symbol == symbol:
            return Epsilon()
        return Empty()

    def __str__(self):
        return self.symbol


class Concatenation(RegEx):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def nullable(self):
        return self.left.nullable() and self.right.nullable()

    def derivative(self, symbol):
```

```python
        left_derivative = self.left.derivative(symbol)

        if isinstance(left_derivative, Empty):
            if self.left.nullable():
                return self.right.derivative(symbol)
            return Empty()

        if self.left.nullable():
            right_derivative = self.right.derivative(symbol)
            if isinstance(right_derivative, Empty):
                return Concatenation(left_derivative, self.right)
            return Alternative(
                Concatenation(left_derivative, self.right), right_derivative
            )
        else:
            return Concatenation(left_derivative, self.right)

    def __str__(self):
        return f"({self.left}{self.right})"


class Alternative(RegEx):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def nullable(self):
        return self.left.nullable() or self.right.nullable()

    def derivative(self, symbol):
        left_derivative = self.left.derivative(symbol)
        right_derivative = self.right.derivative(symbol)

        if isinstance(left_derivative, Empty):
            return right_derivative
        if isinstance(right_derivative, Empty):
            return left_derivative

        return Alternative(left_derivative, right_derivative)

    def __str__(self):
        return f"({self.left}|{self.right})"


class KleeneStar(RegEx):
    def __init__(self, expression):
        self.expression = expression
```

```python
    def nullable(self):
        return True

    def derivative(self, symbol):
        derivative = self.expression.derivative(symbol)

        if isinstance(derivative, Empty):
            return Empty()

        return Concatenation(derivative, self)

    def __str__(self):
        return f"({self.expression})*"


class DFA:
    def __init__(self, states, alphabet, transitions, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions
        self.start_state = start_state
        self.accept_states = accept_states

    def accepts(self, string):
        """Check if the DFA accepts the given string."""
        current_state = self.start_state

        for i, symbol in enumerate(string):
            if symbol not in self.alphabet:
                return False

            if (current_state, symbol) not in self.transitions:
                return False

            current_state = self.transitions[(current_state, symbol)]

        return current_state in self.accept_states

    def __str__(self):
        result = "DFA:\n"
        result += f"  States: {self.states}\n"
        result += f"  Alphabet: {self.alphabet}\n"
        result += f"  Start State: {self.start_state}\n"
        result += f"  Accept States: {self.accept_states}\n"
        result += "  Transitions:\n"
```

```
        for (state, symbol), next_state in sorted(self.transitions.items()):
            result += f"    {state} --{symbol}--> {next_state}\n"
        return result


def simplify(regex):
    """
    Simplify regex expressions to canonical form to improve state␣
 ↪identification.
    """
    if (
        isinstance(regex, Empty)
        or isinstance(regex, Epsilon)
        or isinstance(regex, Symbol)
    ):
        return regex

    # For alternatives
    if isinstance(regex, Alternative):
        left = simplify(regex.left)
        right = simplify(regex.right)

        if isinstance(left, Empty):
            return right
        if isinstance(right, Empty):
            return left

        if str(left) == str(right):
            return left

        return Alternative(left, right)

    # For concatenations
    if isinstance(regex, Concatenation):
        left = simplify(regex.left)
        right = simplify(regex.right)

        if isinstance(left, Empty) or isinstance(right, Empty):
            return Empty()

        if isinstance(left, Epsilon):
            return right

        if isinstance(right, Epsilon):
            return left

        return Concatenation(left, right)
```

```python
    # For Kleene star
    if isinstance(regex, KleeneStar):
        inner = simplify(regex.expression)

        if isinstance(inner, KleeneStar):
            return inner

        if isinstance(inner, Epsilon):
            return Epsilon()

        if isinstance(inner, Empty):
            return Epsilon()

        return KleeneStar(inner)

    return regex


def build_dfa(regex: RegEx, alphabet: set[str]) -> Optional[DFA]:
    # Initialize data structures
    states = set()  # Set of state names (q0, q1, etc.)
    state_to_regex = {}  # Maps state names to their regex
    accept_states = set()  # Set of accepting state names
    transitions = {}  # Maps (state, symbol) pairs to next state
    regex_to_state = {}  # Maps string representations of regex to state names

    # Initialize state counter for generating unique state names
    state_counter = 0

    def new_state():
        nonlocal state_counter
        state_name = f"q{state_counter}"
        state_counter += 1
        return state_name

    # YOUR CODE HERE
    # TODO: Implement the Brzozowski algorithm to convert regex to DFA
    # Steps:
    # 1. Start with the initial regex as the start state
    start_state = new_state()
    states.add(start_state)

    state_to_regex[start_state] = regex
    regex_to_state[regex] = start_state

    if regex.nullable():
```

```
        accept_states.add(start_state)

    # 2. For each state and each symbol in the alphabet:
    #    - Compute the derivative of the state's regex with respect to the␣
↪symbol
    #    - Simplify the resulting regex
    #    - Add a transition from the current state to a state representing this␣
↪new regex
    Q = deque([start_state])

    while Q:
        current_state = Q.popleft()
        current_regex = state_to_regex[current_state]

        for symbol in alphabet:
            d = current_regex.derivative(symbol)
            d = simplify(d)

            if d in regex_to_state:
                next_state = regex_to_state[d]
            else:
                next_state = new_state()
                states.add(next_state)
                state_to_regex[next_state] = d
                regex_to_state[d] = next_state
                Q.append(next_state)

                if d.nullable():
                    accept_states.add(next_state)

            transitions[(current_state, symbol)] = next_state

    # 3. States are accepting if their regex is nullable
    # 4. Continue until no new states are discovered

    # Return the constructed DFA
    # You should return DFA(states, alphabet, transitions, start_state,␣
↪accept_states)
    return DFA(states, alphabet, transitions, start_state, accept_states)
```

4.1 Opis do zadania 4

Każdy rozważany stan ma unikatową nazwę oznaczoną jako q_i, deklarujemy także mapy, które pozwalają nam jednoznacznie dowiedzieć się, która nazwa stanu odnosi się do konkretnego stanu i vice versa. Następnie budujmy DFA zgodnie z algorytmem Brzozowskiego, rozważając pochodne Brzozowskiego dla danego regexa. Sprawdzamy czy w naszych mapach istnieje już zapis nazwy stanu odpowiadającej konkretnemu regexowi: jeśli tak to ten zapis staje się następnym stanem, a jeśli nie to tworzymy nowy zapis, dodajemy elementy do map i aktualizujemy kolejkę. Jeśli dany regex jest nullable to zapisujemy go do zbioru stanów akceptowanych przez DFA. Dla każdego symbolu i każdego stanu zapisujemy odpowiednie przejścia w mapie transitions.

Operacje sprawdzania czy regex jest nullable oraz wyznaczania pochodnych Brzozowskiego są oparte na wcześniej przygotowanych strukturach.