Name: Siddhi Kothekar

Roll No.19

Batch : A2

Practical No. 4

Aim :

| from collections import defaultdict from tabulate import tabulate  def pre\_process(grammer): #Take epsilon as "ε" terminals = [] non\_terminals = [] for i in grammer: if i[0] in terminals: terminals.remove(i[0]) non\_terminals.append(i[0]) else:  non\_terminals.append(i[0]) for j in i[2:]: if j == '|' or j == '' or j == "ε": continue if j in terminals or j in non\_terminals:  continue else: terminals.append(j) return non\_terminals, terminals    def contains\_epsilon(expression):  return "ε" in expression |
| --- |

1. Write a program to validate a string using the parsing table generated in practical 3. Consider valid &amp; invalid string &amp; print the stack &amp; buffer input tracing.

Input: LL (1) parsing table

Implementation: String parsing rules using stack

Output: Each step-in string parsing and whether the input string is valid or invalid.

Code:

| def fetch\_from\_grammar(non\_terminal, grammar): for rule in grammar: lhs, rhs = rule.split(":") if lhs == non\_terminal: return rhs.split("|"), lhs, rhs return ["Error fetching productions"]    def first(expression, terminals, non\_terminals, grammar, first\_dict): if expression in first\_dict: # Use memoization return first\_dict[expression]    first\_set = set()    # Split into alternatives (for cases like A → α | β)  productions, \_, \_ = fetch\_from\_grammar(expression, grammar) if expression in non\_terminals else [expression]  for production in productions: if production == "ε": # Epsilon production  first\_set.add("ε") elif production[0] in terminals: first\_set.add(production[0]) else:  for symbol in production: sub\_first = first(symbol, terminals, non\_terminals, grammar, first\_dict) first\_set.update(sub\_first - {"ε"}) # Add everything except epsilon  if "ε" not in sub\_first:  break # Stop if epsilon is not present else: first\_set.add("ε") # If all symbols produce epsilon, add epsilon    first\_dict[expression] = first\_set return first\_set    def calculate\_first(grammar, terminals, non\_terminals):    # print(f"Terminals: {terminals}") |
| --- |

| # print(f"Non-terminals: {non\_terminals}") first\_dict = {}  for nt in non\_terminals: first\_dict[nt] = first(nt, terminals, non\_terminals, grammar, first\_dict) return first\_dict  def follow(non\_terminal, grammar, terminals, non\_terminals, first\_dict, follow\_dict): if non\_terminal in follow\_dict: return follow\_dict[non\_terminal] # Memoization    follow\_set = set()  if non\_terminal == non\_terminals[0]: # Start symbol follow\_set.add("$") # End of input marker  for rule in grammar: productions, lhs, \_ = fetch\_from\_grammar(rule[0], grammar) for production in productions: indices = [i for i, symbol in enumerate(production) if symbol == non\_terminal] for index in indices: if index + 1 < len(production): # Symbol is not at the end next\_symbol = production[index + 1] if next\_symbol in terminals:  follow\_set.add(next\_symbol) else: first\_next = first\_dict[next\_symbol] - {"ε"} follow\_set.update(first\_next)  if "ε" in first\_dict[next\_symbol]: #case 3  follow\_set.update(follow(lhs, grammar, terminals, non\_terminals, first\_dict, follow\_dict))  else: # Symbol is at the end if lhs != non\_terminal: follow\_set.update(follow(lhs, grammar, terminals, non\_terminals, first\_dict, follow\_dict))    follow\_dict[non\_terminal] = follow\_set return follow\_set |
| --- |

| def calculate\_follow(grammar, terminals, non\_terminals, first\_dict):    follow\_dict = {}  for nt in non\_terminals: follow\_dict[nt] = follow(nt, grammar, terminals, non\_terminals, first\_dict, follow\_dict)    return follow\_dict  def construct\_ll1\_table(grammar, terminals, first, follow): table = defaultdict(dict) conflicts = False  for rule in grammar:  lhs, rhs = rule.split(":")  rhs = rhs.replace(" ", "") # Normalize spaces productions = rhs.split("|")  for production in productions: first\_set = set() if production == "ε": first\_set.add("ε") else:  for symbol in production: if symbol in terminals: first\_set.add(symbol) break  first\_set |= first[symbol] - {"ε"} if "ε" not in first[symbol]:  break else: first\_set.add("ε")  for terminal in first\_set: if terminal != "ε":  if terminal in table[lhs]:  print(f"Conflict detected at M[{lhs}, {terminal}]!") conflicts = True |
| --- |

table[lhs][terminal] = f"{lhs} → {production}"

| if "ε" in first\_set: # Check if production can derive epsilon  for terminal in follow.get(lhs, []): if terminal in table[lhs]: print(f"Conflict detected at M[{lhs}, {terminal}]!") conflicts = True  table[lhs][terminal] = f"{lhs} → ε"    return table, conflicts  def print\_ll1\_table(table, first, follow, conflicts): terminals = sorted(set(t for rules in table.values() for t in rules) | set(term for nt in follow for term in follow[nt])) non\_terminals = sorted(table.keys())  if "$" not in terminals:  terminals.append("$")    headers = ["Non-Terminal"] + terminals rows = []  for non\_terminal in non\_terminals: row = [non\_terminal] + [table[non\_terminal].get(term, "-") for term in terminals] rows.append(row)    print("\nLL(1) Parsing Table:")  print(tabulate(rows, headers, tablefmt="grid")) if conflicts:  print("\n The grammar is NOT LL(1) due to conflicts.") else:  print("\n The grammar is LL(1).")  def parse\_string(parsing\_table, start\_symbol, input\_string):  stack = ["$", start\_symbol] buffer = list(input\_string) + ["$"]    print("\nParsing steps:")  print(f"{'Stack':<20} {'Buffer':<20} {'Action'}") |
| --- |

| stack\_top = stack[-1]  buffer\_top = buffer[0]    print(f"{' '.join(stack):<20} {' '.join(buffer):<20}", end=" ")  if stack\_top == buffer\_top == "$":  print("String is valid.") return True  elif stack\_top == buffer\_top: stack.pop() buffer.pop(0)  print(f"Matched '{stack\_top}'")  elif stack\_top in parsing\_table: if buffer\_top in parsing\_table[stack\_top]: production = parsing\_table[stack\_top][buffer\_top] stack.pop()    # Extract the right-hand side of the production rhs = production.split("→")[1].strip() if rhs != "ε":  symbols = list(rhs) # Split into individual characters stack.extend(reversed(symbols))  print(f"Use rule ({stack\_top}, {buffer\_top}), {production}") else:  print(f"No rule for ({stack\_top}, {buffer\_top})") return False  else: print(f"Unexpected symbol '{buffer\_top}'") return False    print("String is invalid.")  return False    # Modified main function to include parsing |
| --- |

while stack: def main():

grammar = ["S:A|BC", "A:a|b", "B:p|ε", "C:c"]

| # grammar = ["A:SB|B", "S:a|Bc|ε", "B:b|d"] non\_terminals, terminals = pre\_process(grammar)  first\_dict = calculate\_first(grammar, terminals, non\_terminals)  follow\_dict = calculate\_follow(grammar, terminals, non\_terminals, first\_dict)    print("FIRST sets:") for non\_terminal, first\_set in first\_dict.items():  print(f"FIRST({non\_terminal}) = {first\_set}")    print("\nFOLLOW sets:") for non\_terminal, follow\_set in follow\_dict.items():  print(f"FOLLOW({non\_terminal}) = {follow\_set}")    ll1\_parsing\_table, conflict = construct\_ll1\_table(grammar, terminals, first\_dict, follow\_dict) print\_ll1\_table(ll1\_parsing\_table, first\_dict, follow\_dict, conflict)  # Add parsing example  if not conflict: # Only try parsing if grammar is LL(1) test\_string = "pc"  print(f"\nAttempting to parse string: {test\_string}") result = parse\_string(ll1\_parsing\_table, "S", test\_string) print(f"Parse result: {'Accepted' if result else 'Rejected'}")  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |

Output:



1. Use Virtual Lab on LL1 parser to validate the string and verify your string validation using simulation.

Link for Virtual Lab:

Link : https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/

Output: Validation from Virtual lab simulator

Go to Virtual lab: Input the grammar, string and validate your string parsing with the simulator

(screen shot expected).

