Practical 3

First:

import re

from collections import defaultdict

def read\_grammar():

"""Reads grammar productions from user input."""

grammar = []

print("Enter the grammar (each production in the form: A -> a | B C | ...), enter 'done' when finished:")

while True:

rule = input("Enter production rule: ").strip()

if rule.lower() == 'done':

break

if "->" not in rule:

print("Invalid format. Use 'A -> B | C'.")

continue

grammar.append(rule)

return grammar

def parse\_grammar(grammar):

"""Parses the grammar into a dictionary."""

rules = defaultdict(list)

for rule in grammar:

left\_side, right\_side = rule.split("->")

left\_side = left\_side.strip()

right\_sides = right\_side.split("|")

for side in right\_sides:

rules[left\_side].append(side.strip().split())

return rules

def get\_first\_of\_right\_side(right\_side, first\_sets, rules):

"""Computes the FIRST set for a given right-hand side."""

first = set()

for symbol in right\_side:

if not symbol.isupper(): # Terminal

first.add(symbol)

break

else: # Non-terminal

first\_of\_symbol = first\_sets[symbol]

first.update(first\_of\_symbol - {'ε'})

if 'ε' not in first\_of\_symbol:

break

else:

first.add('ε') # If all symbols derive ε

return first

def find\_first\_sets(grammar):

"""Computes FIRST sets for all non-terminals."""

first\_sets = defaultdict(set)

rules = parse\_grammar(grammar)

while True:

changes = False

for left\_side, right\_sides in rules.items():

for right\_side in right\_sides:

first\_of\_right = get\_first\_of\_right\_side(right\_side, first\_sets, rules)

if not first\_of\_right.issubset(first\_sets[left\_side]):

first\_sets[left\_side].update(first\_of\_right)

changes = True

if not changes:

break

return first\_sets

def print\_first\_sets(first\_sets):

"""Prints the FIRST sets for all non-terminals."""

print("\nFIRST sets:")

for non\_terminal, first in first\_sets.items():

print(f"FIRST({non\_terminal}) = {first}")

def main():

grammar = read\_grammar()

if not grammar:

print("No grammar entered. Exiting.")

return

first\_sets = find\_first\_sets(grammar)

print\_first\_sets(first\_sets)

if \_\_name\_\_ == "\_\_main\_\_":

main()

LL PARSER

def read\_grammar():

    """

    Reads grammar productions from user input.

    Each line should be of the form:

      NonTerminal -> production1 | production2 | ...

    where each production is either space-separated tokens or a single string

(each character is a symbol).

    Enter an empty line to finish.

    """

    grammar = {}

    print("Enter grammar productions one per line in the format:")

    print("  NonTerminal -> production1 | production2 | ...")

    print("Use 'e' to denote the empty production. Enter an empty line when done.")

    while True:

        line = input().strip()

        if not line:

            break

        if "->" not in line:

            print("Invalid format. Please include '->'.")

            continue

        lhs, rhs = line.split("->")

        lhs = lhs.strip()

        # Initialize production list for lhs if not already present

        if lhs not in grammar:

            grammar[lhs] = []

        # Split the right-hand side into alternatives

        alternatives = rhs.split("|")

        for alt in alternatives:

            alt = alt.strip()

            # If the production is exactly "ε", use it as a marker for empty production.

            if alt == "ε":

                grammar[lhs].append(["ε"])

            else:

                # If there are spaces, split on whitespace; otherwise, treat each character as a symbol.

                if " " in alt:

                    tokens = alt.split()

                else:

                    tokens = list(alt)

                grammar[lhs].append(tokens)

    return grammar # This line was likely indented incorrectly. Make sure it aligns with the 'def' keyword.

def compute\_first\_sets(grammar):

    """

    Computes the FIRST sets for all nonterminals in the grammar.

    Productions are lists of tokens. Terminals are assumed to be symbols not in

grammar.

    """

    first = { nonterm: set() for nonterm in grammar }

    changed = True

    while changed:

        changed = False

        for nonterm, productions in grammar.items():

            for prod in productions:

                # If production is ['ε'], add ε

                if prod == ["ε"]:

                    if "ε" not in first[nonterm]:

                        first[nonterm].add("ε")

                        changed = True

                else:

                    for symbol in prod:

                        # symbol is terminal if not a nonterminal (i.e. not in grammar) or is literally 'ε'

                        if symbol not in grammar:

                            if symbol not in first[nonterm]:

                                first[nonterm].add(symbol)

                                changed = True

                            break

                        else:

                            # symbol is nonterminal: add all of its FIRST except ε.

                            before = len(first[nonterm])

                            first[nonterm].update(first[symbol] - {"ε"})

                            if len(first[nonterm]) > before:

                                changed = True

                            if "ε" in first[symbol]:

                                # If ε is in FIRST(symbol) and symbol is the lasttoken,

                                # then add ε to FIRST(nonterm)

                                if symbol == prod[-1]:

                                    if "ε" not in first[nonterm]:

                                        first[nonterm].add("ε")

                                        changed = True

                                continue

                            else:

                                break

    return first

def first\_of\_string(symbols, first, grammar):

    """

    Compute FIRST for a sequence of symbols.

    """

    result = set()

    for symbol in symbols:

        if symbol not in grammar:

            result.add(symbol)

            return result

        else:

            result.update(first[symbol] - {"ε"})

            if "ε" in first[symbol]:

                if symbol == symbols[-1]:

                    result.add("ε")

                continue

            else:

                return result

    return result

def compute\_follow\_sets(grammar, first, start\_symbol):

    """

    Computes FOLLOW sets for all nonterminals in the grammar.

    """

    follow = { nonterm: set() for nonterm in grammar }

    follow[start\_symbol].add("$")  # End-of-input marker

    changed = True

    while changed:

        changed = False

        for A, productions in grammar.items():

            for prod in productions:

                for i, symbol in enumerate(prod):

                    if symbol in grammar:  # Only nonterminals get a FOLLOW set.

                        follow\_before = len(follow[symbol])

                        beta = prod[i+1:]

                        if beta:

                            beta\_first = first\_of\_string(beta, first, grammar)

                            follow[symbol].update(beta\_first - {"ε"})

                            if "ε" in beta\_first:

                                follow[symbol].update(follow[A])

                        else:

                            follow[symbol].update(follow[A])

                        if len(follow[symbol]) > follow\_before:

                            changed = True

    return follow

def construct\_ll1\_table(grammar, first, follow):

    """

    Constructs the LL(1) parsing table.

    The table is a dictionary mapping nonterminals to a dictionary that maps

terminals to a production.

    """

    table = { nonterm: {} for nonterm in grammar }

    def add\_entry(nonterm, terminal, production):

        if terminal in table[nonterm]:

            print(f"Conflict detected for table[{nonterm}, {terminal}]. Grammar may not be LL(1).")

        table[nonterm][terminal] = production

    for nonterm, productions in grammar.items():

        for prod in productions:

            prod\_first = first\_of\_string(prod, first, grammar)

            for terminal in prod\_first - {"ε"}:

                add\_entry(nonterm, terminal, prod)

            if "ε" in prod\_first:

                for terminal in follow[nonterm]:

                    add\_entry(nonterm, terminal, prod)

    return table

def parse\_input\_string(input\_string, start\_symbol, table, grammar):

    """

    Parse the input string using the LL(1) parsing table.

    This function assumes that tokens are separated by spaces.

    """

    tokens = input\_string.split() + ["$"]

    stack = ["$", start\_symbol]

    index = 0

    while stack:

        top = stack.pop()

        current = tokens[index]

        if top not in grammar and top != "$":  # Terminal symbol

            if top == current:

                index += 1

            else:

                print(f"Error: expected {top} but got {current}")

                return False

        elif top == "$":

            if top == current:

                return True

            else:

                print("Error at end of parsing.")

                return False

        else:

            # top is nonterminal; consult table

            if current in table[top]:

                production = table[top][current]

                if production != ["ε"]:

                    for sym in reversed(production):

                        stack.append(sym)

            else:

                print(f"No rule for nonterminal {top} with token {current}")

                return False

    return False

def main():

    # Read grammar from project input.

    grammar = read\_grammar()

    if not grammar:

        print("No grammar entered. Exiting.")

        return

    # Assume the first entered nonterminal is the start symbol.

    start\_symbol = list(grammar.keys())[0]

    print("\nGrammar:")

    for nt, prods in grammar.items():

        for prod in prods:

            print(f"  {nt} -> {' '.join(prod)}")

    # Compute FIRST and FOLLOW sets.

    first = compute\_first\_sets(grammar)

    follow = compute\_follow\_sets(grammar, first, start\_symbol)

    print("\nFIRST sets:")

    for nt in grammar:

        print(f"  {nt}: {first[nt]}")

    print("\nFOLLOW sets:")

    for nt in grammar:

        print(f"  {nt}: {follow[nt]}")

    # Construct LL(1) parsing table.

    table = construct\_ll1\_table(grammar, first, follow)

    print("\nLL(1) Parsing Table:")

    for nt, rules in table.items():

        for terminal, production in rules.items():

            print(f"  M[{nt}, {terminal}] = {nt} -> {' '.join(production)}")

    # Optionally, parse an input string.

    print("\nEnter an input string to parse (tokens separated by space), or press Enter to skip:")

    input\_string = input().strip()

    if input\_string:

        if parse\_input\_string(input\_string, start\_symbol, table, grammar):

            print("Input is accepted by the grammar.")

        else:

            print("Input is not accepted by the grammar.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()