LAB SESSION 6: PREDICTIVE PARSER USING PYTHON

AIM: To implement a Predictive parser using Python.

PROBLEM DEFINITION: Develop a python program to implement predictive parser for a given grammar.

THEORY: Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar.
The goal of the parser is to determine the syntactic validity of a source string is valid; a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue a diagnostic message identifying the nature and cause of the errors in the string. Every elementary subtree in the parse tree corresponds to a production of the grammar.
There are two ways of identifying an elementary subtree:
1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal
Predictive Parser is also another method that implements the technique of Top- Down parsing without Backtracking. A predictive parser is an effective technique of executing recursive-descent parsing by managing the stack of activation records, particularly.
How predictive parsers work
LL(1) Parsing: The parser reads the input from left to right, a leftmost derivation is used to construct the parse tree, and the decision for the next step is based on a single look-ahead token.
Stack: A stack is used to store grammar symbols. It is initialized with the grammar's start symbol and a special end-of-input marker.
Parsing Table: A 2D table is used to guide the parser's decisions. The table is indexed by a non-terminal (from the top of the stack) and the current input token.
No Backtracking: A predictive parser does not need to backtrack because the parsing table contains a unique entry for each combination of non-terminal and input token, ensuring a single, deterministic path.

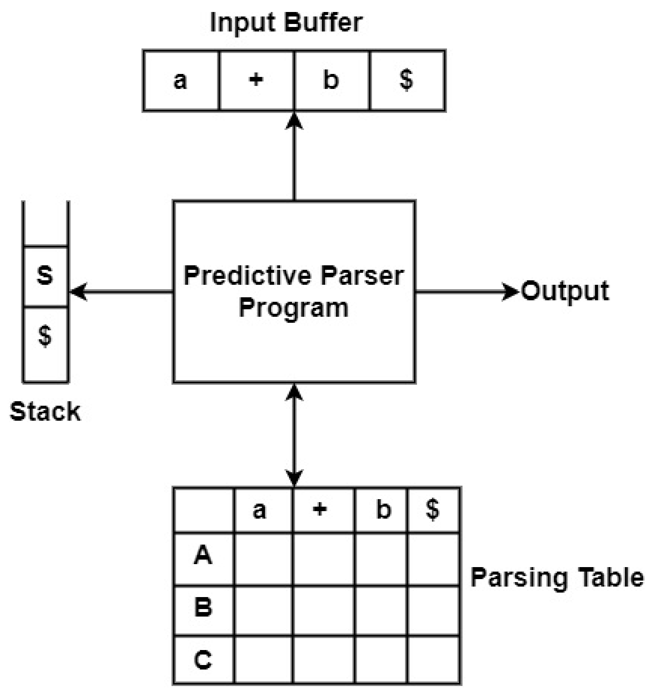
Key Components:

1. **Input Buffer:** The input buffer includes the string to be parsed followed by an end marker $ to denote the end of the string.

a	+	b	$

Input String

2. **Stack** – It contains a combination of grammar symbols with $ on the bottom of the stack. At the start of Parsing, the stack contains the start symbol of Grammar followed by $.



Predictive Parser

3. **Parsing Table** – It is a two-dimensional array or Matrix M [A, a] where A is nonterminal and 'a' is a terminal symbol.

Following are the steps to perform Predictive Parsing

1. Elimination of Left Recursion
2. Left Factoring
3. Computation of FIRST & FOLLOW
4. Construction of Predictive Parsing Table
5. Parse the Input String

Algorithm to construct Predictive Parsing Table

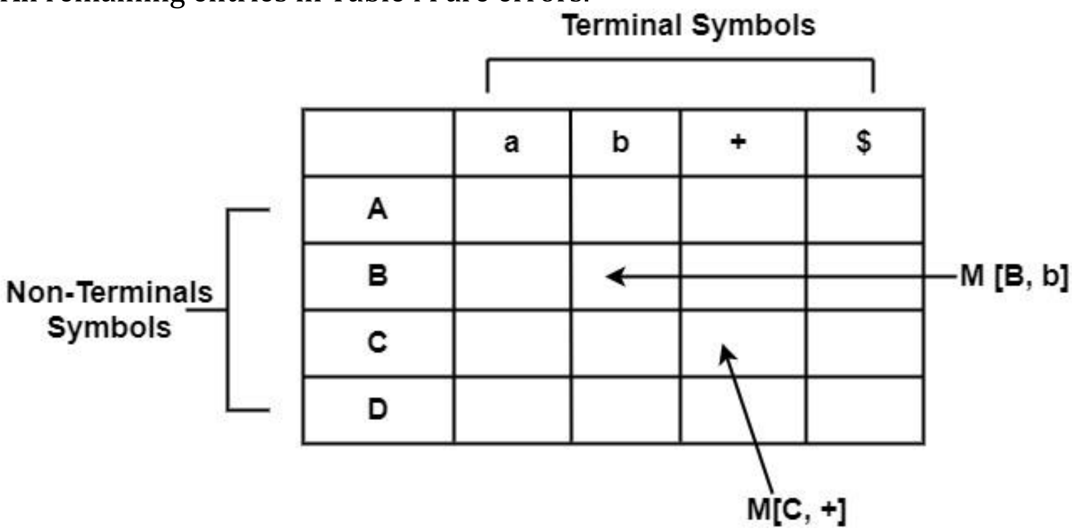
Input – Context-Free Grammar G

Output – Predictive Parsing Table M

Method – For the production A → α of Grammar G.

- For each terminal, a in FIRST (?) add A → α to M [A, a].
- If ε is in FIRST (α), and b is in FOLLOW (A), then add A → α to M[A, b].
- If ε is in FIRST (α), and $ is in FOLLOW (A), then add A → α to M[A, $].

- All remaining entries in Table M are errors.



PROGRAM:

```
# Predictive Parser Implementation in Python
PROD_ARROW = '⇒'  # arrow used when printing productions

# Function to compute FIRST sets
def compute_first(symbol, productions, first):
    # If it's the epsilon symbol
    if symbol == 'ε':
        return {'ε'}
    # If symbol is not a non-terminal (i.e. not a key in productions), it's a terminal
    if symbol not in productions:
        return {symbol}
    result = set()
    for prod in productions.get(symbol, []):
        # prod is a list of symbols
        if len(prod) == 1 and prod[0] == 'ε':
            result.add('ε')
        else:
            for s in prod:
                temp = compute_first(s, productions, first)
                result |= (temp - {'ε'})
                if 'ε' not in temp:
                    break
            else:
                result.add('ε')
    return result
```

```
def compute_all_firsts(productions):
    """Compute FIRST sets for all non-terminals using an iterative fixpoint algorithm.

    This is safer than naive recursion for grammars with left recursion.
    """
    first = {nt: set() for nt in productions}
    changed = True
    while changed:
        changed = False
        for head, prods in productions.items():
            for prod in prods:
                # epsilon production
                if len(prod) == 1 and prod[0] == 'ε':
                    if 'ε' not in first[head]:
                        first[head].add('ε')
                        changed = True
                    continue

                # walk symbols in RHS
                all_eps = True
                for sym in prod:
                    if sym not in productions:
                        # terminal
                        if sym not in first[head]:
                            first[head].add(sym)
                            changed = True
                        all_eps = False
                        break
                    else:
                        # non-terminal: add FIRST(sym) minus epsilon
                        before = len(first[head])
                        to_add = first[sym] - {'ε'}
                        if to_add - first[head]:
                            first[head] |= to_add
                            changed = True
                        if 'ε' in first[sym]:
                            # sym can produce epsilon; continue to next symbol
                            continue
                        else:
                            all_eps = False
                            break
```

```
            if all_eps:
                # all symbols can derive epsilon
                if 'ε' not in first[head]:
                    first[head].add('ε')
                    changed = True

    return first

# Function to compute FOLLOW sets
def compute_follow(symbol, productions, start_symbol, first, follow):
    if symbol not in follow:
        follow[symbol] = set()
    if symbol == start_symbol:
        follow[symbol].add('$')
    for head, prods in productions.items():
        for prod in prods:
            for i, s in enumerate(prod):
                if s == symbol:
                    rest = prod[i+1:]
                    temp = set()
                    if rest:
                        for r in rest:
                            # if r has a FIRST set use it; otherwise r is a terminal
                            if r in first:
                                temp |= (first[r] - {'ε'})
                                if 'ε' not in first[r]:
                                    break
                            else:
                                temp |= {r}
                                break
                        else:
                            temp |= follow.get(head, set())
                    else:
                        temp |= follow.get(head, set())
                    follow[symbol] |= temp
    return follow

# Function to construct predictive parsing table
def construct_table(productions, first, follow):
    table = {}
    origins = {}  # map (A, t) -> "A -> ..." string that added the entry
```

```
    conflicts = []
    for head, prods in productions.items():
        for prod in prods:
            prod_str = ' '.join(prod)
            first_set = set()
            if not (len(prod) == 1 and prod[0] == 'ε'):
                for s in prod:
                    if s in first:
                        first_set |= (first[s] - {'ε'})
                        if 'ε' not in first[s]:
                            break
                    else:
                        # s is a terminal
                        first_set |= {s}
                        break
                else:
                    first_set.add('ε')
            else:
                first_set.add('ε')

            for terminal in first_set - {'ε'}:
                key = (head, terminal)
                if key in table and table[key] != prod:
                    conflicts.append((head, terminal, table[key], origins.get(key, ''), prod,
f"{head} {PROD_ARROW} {prod_str}"))
                else:
                    table[key] = prod
                    origins[key] = f"{head} {PROD_ARROW} {prod_str}"

            if 'ε' in first_set:
                for terminal in follow.get(head, set()):
                    key = (head, terminal)
                    if key in table and table[key] != prod:
                        conflicts.append((head, terminal, table[key], origins.get(key, ''), prod,
f"{head} {PROD_ARROW} {prod_str}"))
                    else:
                        table[key] = prod
                        origins[key] = f"{head} {PROD_ARROW} {prod_str}"

    return table, conflicts, origins

def compute_all_follows(productions, start_symbol, first):
```

```
    """Compute FOLLOW sets using an iterative fixpoint algorithm."""
    follow = {nt: set() for nt in productions}
    follow[start_symbol].add('$')
    changed = True
    while changed:
        changed = False
        for head, prods in productions.items():
            for prod in prods:
                for i, B in enumerate(prod):
                    if B not in productions:
                        continue
                    # rest of symbols after B
                    rest = prod[i+1:]
                    # compute FIRST(rest)
                    first_rest = set()
                    if not rest:
                        # add follow(head) to follow(B)
                        before = len(follow[B])
                        follow[B] |= follow[head]
                        if len(follow[B]) != before:
                            changed = True
                    else:
                        add_follow_head = True
                        for sym in rest:
                            if sym in first:
                                before = len(follow[B])
                                follow[B] |= (first[sym] - {'ε'})
                                if len(follow[B]) != before:
                                    changed = True
                                if 'ε' in first[sym]:
                                    # continue to next symbol
                                    continue
                                else:
                                    add_follow_head = False
                                    break
                            else:
                                # sym is terminal
                                before = len(follow[B])
                                follow[B].add(sym)
                                if len(follow[B]) != before:
                                    changed = True
                                add_follow_head = False
```

```
                    break
                if add_follow_head:
                    before = len(follow[B])
                    follow[B] |= follow[head]
                    if len(follow[B]) != before:
                        changed = True
    return follow

def terminals_from_productions(productions):
    terms = set()
    nonterms = set(productions.keys())
    for rhs in productions.values():
        for prod in rhs:
            for sym in prod:
                if sym == 'ε':
                    continue
                if sym not in nonterms:
                    terms.add(sym)
    return sorted(terms)

def pretty_sym(sym):
    # Print symbols as-is; epsilon remains the character 'ε'
    return sym

def print_firsts_and_follows(productions, first, follow):
    nonterms = list(productions.keys())
    print('\nCalculated firsts:')
    for nt in nonterms:
        print(f"first({nt}) => {{{', '.join(sorted(first.get(nt, set())) )}}}")

    print('\nCalculated follows:')
    for nt in nonterms:
        print(f"follow({nt}) => {{{', '.join(sorted(follow.get(nt, set())) )}}}")

    # Nicely formatted table
    print('\nFirsts and Follow Result table\n')
    # prepare column widths
    col1 = max(6, max(len(nt) for nt in nonterms)) + 2
    col2 = max(10, max(len(str(sorted(first.get(nt, set())))) for nt in nonterms) ) + 4
    col3 = max(10, max(len(str(sorted(follow.get(nt, set())))) for nt in nonterms) ) + 4
    # header
    print(f"{ 'Non-T':<{col1}}{ 'FIRST':<{col2}}{ 'FOLLOW':<{col3}}")
```

```
    for nt in nonterms:
        fset = sorted(first.get(nt, set()))
        foset = sorted(follow.get(nt, set()))
        print(f"{nt:<{col1}}{str(fset):<{col2}}{str(foset):<{col3}}")

def print_parsing_table(productions, table):
    nonterms = list(productions.keys())
    terms = terminals_from_productions(productions)
    # include $ and ensure unique & keep order
    if '$' not in terms:
        terms = terms + ['$']
    # compute column widths based on content
    col_widths = {}
    # header widths
    col_widths['nonterm'] = max(6, max(len(nt) for nt in nonterms)) + 2
    for t in terms:
        max_cell = len(t)
        for A in nonterms:
            if (A, t) in table:
                prod = table[(A, t)]
                cell = f"{A} {PROD_ARROW} {' '.join(pretty_sym(s) for s in prod)}"
                max_cell = max(max_cell, len(cell))
        col_widths[t] = max_cell + 2

    # print header
    header = f"{'' :<{col_widths['nonterm']}}"
    for t in terms:
        header += f"{t:<{col_widths[t]}}"
    print('\nGenerated parsing table:\n')
    print(header)
    # print rows
    for A in nonterms:
        row = f"{A:<{col_widths['nonterm']}}"
        for t in terms:
            cell = ''
            if (A, t) in table:
                prod = table[(A, t)]
                cell = f"{A} {PROD_ARROW} {' '.join(pretty_sym(s) for s in prod)}"
            row += f"{cell:<{col_widths[t]}}"
        print(row)

# Parsing function
```

```
def predictive_parse(input_string, start_symbol, table):
    # Tokenize input string into grammar tokens (space separated tokens expected)
    tokens = tokenize(input_string)
    tokens.append('$')
    stack = ['$']
    stack.append(start_symbol)
    i = 0

    print(f"{'Buffer':<30}{'Stack':<30}{'Action'}")
    while True:
        buffer_str = ' '.join(tokens[i:])
        # display stack with top on the left (reverse of internal list)
        stack_str = ' '.join(reversed(stack))
        # peek top
        top = stack.pop() if stack else None
        current_input = tokens[i] if i < len(tokens) else '$'
        action = ''
        if top == current_input == '$':
            print(f"{buffer_str:<30}{stack_str:<30}{'Accept'}")
            return True
        elif top == current_input:
            action = f"Matched:{current_input}"
            print(f"{buffer_str:<30}{stack_str:<30}{action}")
            i += 1
        elif (top, current_input) in table:
            prod = table[(top, current_input)]
                action = f"T[{top}][{current_input}] = {top} {PROD_ARROW} {'
'.join(pretty_sym(s) for s in prod)}"
            print(f"{buffer_str:<30}{stack_str:<30}{action}")
            # push RHS in reverse (unless epsilon)
            if not (len(prod) == 1 and prod[0] == 'ε'):
                for symbol in reversed(prod):
                    stack.append(symbol)
            # after pushing, show updated stack state
            new_stack_str = ' '.join(reversed(stack))
            print(f"{buffer_str:<30}{new_stack_str:<30}{''}")
        else:
            action = 'Error: no rule'
            print(f"{buffer_str:<30}{stack_str:<30}{action}")
            return False
```

```
def tokenize(s):
    """Very small tokenizer for the sample grammar: recognizes 'id', operators and parentheses."""
    tokens = []
    i = 0
    while i < len(s):
        if s[i].isspace():
            i += 1
            continue
        if s.startswith('id', i):
            tokens.append('id')
            i += 2
            continue
        # single char tokens
        if s[i] in ['+', '*', '(', ')']:
            tokens.append(s[i])
            i += 1
            continue
        # unknown sequence (collect as a single token)
        j = i
        while j < len(s) and not s[j].isspace() and s[j] not in ['+', '*', '(', ')']:
            j += 1
        tokens.append(s[i:j])
        i = j
    return tokens

import argparse
import sys

def load_grammar(path):
    """Load grammar from a file.

    Expected simple format (examples present in grammar.txt):
    - Lines starting with # are comments
    - Start: S        (optional; overrides first non-terminal)
    - Input: b a      (optional; input tokens separated by spaces)
    - A -> a b | c    (productions)
    """
    productions = {}
    start_symbol = None
    input_tokens = None
    with open(path, 'r', encoding='utf-8') as f:
```

```
        for raw in f:
            line = raw.strip()
            if not line or line.startswith('#'):
                continue
            if line.lower().startswith('start:'):
                start_symbol = line.split(':', 1)[1].strip()
                continue
            if line.lower().startswith('input:'):
                input_tokens = line.split(':', 1)[1].strip()
                continue
            if '->' in line:
                head, rhs = line.split('->', 1)
                head = head.strip()
                alternatives = [alt.strip() for alt in rhs.split('|')]
                prods = []
                for alt in alternatives:
                    if alt == '' or alt == 'ε' or alt.lower() == 'eps':
                        prods.append(['ε'])
                    else:
                        tokens = alt.split()
                        prods.append(tokens)
                productions.setdefault(head, []).extend(prods)
    # If no start symbol provided, pick first LHS
    if not start_symbol:
        if productions:
            start_symbol = next(iter(productions))
    return productions, start_symbol, input_tokens

def load_multi_tests(path):
    """Load multiple testcases from a consolidated file.

    Block format:
      Test: name
      Start: S
      <productions>
      Valid: <tokens>
      Invalid: <tokens>

    Returns a list of dicts with keys: name, productions, start, valid, invalid
    """
    tests = []
    current = None
```

```
def ensure_current():
    nonlocal current
    if current is None:
        current = {
            'name': None,
            'productions': {},
            'start': None,
            'valid': None,
            'invalid': None,
        }

with open(path, 'r', encoding='utf-8') as f:
    for raw in f:
        line = raw.strip()
        if not line:
            # blank line: if we have a populated block, finalize it
            if current and current['productions'] and current['start']:
                tests.append(current)
                current = None
            continue
        if line.startswith('#'):
            continue
        if line.lower().startswith('test:'):
            # finalize previous test if any
            if current and current['productions'] and current['start']:
                tests.append(current)
            current = {
                'name': line.split(':', 1)[1].strip(),
                'productions': {},
                'start': None,
                'valid': None,
                'invalid': None,
            }
            continue
        ensure_current()
        if line.lower().startswith('start:'):
            current['start'] = line.split(':', 1)[1].strip()
            continue
        if line.lower().startswith('valid:'):
            current['valid'] = line.split(':', 1)[1].strip()
            continue
        if line.lower().startswith('invalid:'):
```

```
            current['invalid'] = line.split(':', 1)[1].strip()
            continue
        if '->' in line:
            head, rhs = line.split('->', 1)
            head = head.strip()
            alternatives = [alt.strip() for alt in rhs.split('|')]
            prods = []
            for alt in alternatives:
                if alt == '' or alt == 'ε' or alt.lower() == 'eps':
                    prods.append(['ε'])
                else:
                    tokens = alt.split()
                    prods.append(tokens)
            current['productions'].setdefault(head, []).extend(prods)

    # finalize last block
    if current and current['productions'] and current['start']:
        tests.append(current)

    return tests

def format_productions(productions):
    """Return a human friendly string of the grammar productions."""
    lines = []
    for head, prods in productions.items():
        alts = [' '.join(p) for p in prods]
        lines.append(f"{head} {PROD_ARROW} {' | '.join(alts)}")
    return '\n'.join(lines)

def remove_left_recursion(productions):
    """Remove left recursion (indirect + direct) from the grammar.

    Returns (new_productions, steps) where steps is a list of human-readable
    descriptions of each change performed.
    """
    steps = []
    # Work on a copy
    prods = {nt: [list(p) for p in rhs] for nt, rhs in productions.items()}
    nonterminals = list(prods.keys())

    def make_new_nt(base):
        # append a prime marker; ensure uniqueness
```

```
        candidate = base + "'"
        while candidate in prods:
            candidate += "'"
        return candidate

    for i, Ai in enumerate(nonterminals):
        # replace Ai -> Aj α where j < i (indirect left recursion elimination)
        for j in range(i):
            Aj = nonterminals[j]
            new_rhs = []
            changed = False
            for prod in prods[Ai]:
                if prod and prod[0] == Aj:
                    # replace Aj γ with β γ for each Aj -> β
                    rest = prod[1:]
                    for beta in prods[Aj]:
                        new_prod = list(beta) + rest
                        new_rhs.append(new_prod)
                        steps.append(f"In {Ai}: replaced {Ai} {PROD_ARROW} {Aj} {' '.join(rest) if rest else ''} with {Ai} {PROD_ARROW} {' '.join(new_prod)} (expanding {Aj} {PROD_ARROW} {' '.join(beta)})")
                    changed = True
                else:
                    new_rhs.append(prod)
            if changed:
                steps.append(f"After expanding {Aj} in {Ai}, {Ai} productions become: {[' '.join(p) for p in new_rhs]}")
                prods[Ai] = new_rhs

        # now remove direct left recursion for Ai
        alpha = []  # productions where Ai -> Ai α
        beta = []   # productions where Ai -> β (not starting with Ai)
        for prod in prods[Ai]:
            if prod and prod[0] == Ai:
                alpha.append(prod[1:])
            else:
                beta.append(prod)

        if alpha:
            Aip = make_new_nt(Ai)
            steps.append(f"Direct left recursion detected in {Ai}. Creating new non-terminal {Aip} and rewriting productions.")
```

```
            # Ai -> beta Aip
            new_Ai_rhs = []
            for b in beta:
                if b == ['ε']:
                    new_Ai_rhs.append([Aip])
                else:
                    new_Ai_rhs.append(list(b) + [Aip])
            prods[Ai] = new_Ai_rhs
            # Aip -> alpha Aip | ε
            prods[Aip] = []
            for a in alpha:
                prods[Aip].append(list(a) + [Aip])
            prods[Aip].append(['ε'])
            steps.append(f"{Ai} rewritten as: {[' '.join(p) for p in prods[Ai]]}")
            steps.append(f"{Aip} productions: {[' '.join(p) for p in prods[Aip]]}")
            # also record the new nonterminal in order list so subsequent iterations can use it
            nonterminals.insert(i+1, Aip)

    return prods, steps

def left_factor(productions):
    """Apply left factoring to the grammar. Returns (new_productions, steps).

    This does a simple factoring: when a non-terminal has two or more
    alternatives that share a common prefix (at least the first symbol),
    it pulls the common prefix into a new non-terminal.
    """
    prods = {nt: [list(p) for p in rhs] for nt, rhs in productions.items()}
    steps = []

    def make_new_nt(base):
        candidate = base + "'"
        while candidate in prods:
            candidate += "'"
        return candidate

    changed = True
    while changed:
        changed = False
        for A, alternatives in list(prods.items()):
            if len(alternatives) < 2:
```

```
            continue
        # group by first symbol
        groups = {}
        for prod in alternatives:
            key = prod[0] if prod else 'ε'
            groups.setdefault(key, []).append(prod)

        for key, group in groups.items():
            if len(group) < 2:
                continue
            # find longest common prefix among these prods
            prefix = []
            for symbols in zip(*group):
                if all(sym == symbols[0] for sym in symbols):
                    prefix.append(symbols[0])
                else:
                    break
            if not prefix:
                continue
            # create new non-terminal
            A_dash = make_new_nt(A)
             steps.append(f"Left factoring on {A}: common prefix {' '.join(prefix)} found;
created {A_dash}.")
            # build new alternatives
            new_A_alts = []
            for prod in alternatives:
                if prod[:len(prefix)] == prefix:
                    # moved under A_dash
                    suffix = prod[len(prefix):]
                    if not suffix:
                        prods.setdefault(A_dash, []).append(['ε'])
                    else:
                        prods.setdefault(A_dash, []).append(suffix)
                else:
                    new_A_alts.append(prod)
            # A -> prefix A_dash | other_alts
            new_A_alts.append(list(prefix) + [A_dash])
            prods[A] = new_A_alts
            changed = True
            break
        if changed:
            break
```

```
    return prods, steps


def read_ops_file(path):
    try:
        with open(path, 'r', encoding='utf-8') as f:
            return f.read()
    except Exception:
        return None

def main(argv=None):
    parser = argparse.ArgumentParser(description='Predictive parser that loads grammar from a file')
    parser.add_argument('--grammar', '-g', default='grammar.txt', help='Path to grammar file or consolidated tests file (default: grammar.txt)')
    parser.add_argument('--input-string', '-s', help='Input string to parse (tokens separated by spaces where appropriate). If omitted uses Input: from grammar file')
    parser.add_argument('--input-file', '-i', help='File that contains an Input: line or plain input string')
    args = parser.parse_args(argv)

    # Detect consolidated tests file by presence of "Test:" or Valid/Invalid lines
    file_text = ''
    try:
        with open(args.grammar, 'r', encoding='utf-8') as _f:
            file_text = _f.read()
    except Exception:
        pass

    is_multi = any(tag in file_text for tag in ['\nTest:', '\nValid:', '\nInvalid:'])

    if is_multi and not args.input_string and not args.input_file:
        # Run all tests in the file and exit with summary
        tests = load_multi_tests(args.grammar)
        if not tests:
            print(f"No tests parsed from {args.grammar}")
            sys.exit(1)

        # Ensure stdout/stderr use UTF-8 so symbols like 'ε' print correctly on Windows
        try:
            sys.stdout.reconfigure(encoding='utf-8')
```

```
    sys.stderr.reconfigure(encoding='utf-8')
except Exception:
    pass

overall = []
for t in tests:
    name = t['name'] or 'unnamed'
    productions = t['productions']
    start_symbol = t['start']

    print('=' * 80)
    print(f"Test: {name}")
    print(f"Start symbol: {start_symbol}")
    print("Original grammar:\n")
    print(format_productions(productions))
    print('\n')

    # Transformations
    productions_lr_removed, lr_steps = remove_left_recursion(productions)
    if lr_steps:
        print("--- Left Recursion Removal Steps ---")
        for s in lr_steps:
            print("-", s)
        print('\nGrammar after left recursion removal:\n')
        print(format_productions(productions_lr_removed))
        print('\n')
    else:
        print("No left recursion detected.\n")

    productions_factored, lf_steps = left_factor(productions_lr_removed)
    if lf_steps:
        print("--- Left Factoring Steps ---")
        for s in lf_steps:
            print("-", s)
        print('\nGrammar after left factoring:\n')
        print(format_productions(productions_factored))
        print('\n')
    else:
        print("No left factoring needed.\n")

    productions = productions_factored
    first = compute_all_firsts(productions)
```

```
        follow = compute_all_follows(productions, start_symbol, first)
        table, conflicts, origins = construct_table(productions, first, follow)

        print_firsts_and_follows(productions, first, follow)
        print_parsing_table(productions, table)
        if conflicts:
            print('\nGrammar is NOT LL(1). Conflicts found in parsing table:')
            for (A, tok, existing_prod, existing_origin, new_prod, new_origin) in conflicts:
                existing_str = existing_origin or (' '.join(existing_prod))
                new_str = new_origin or (' '.join(new_prod))
                    print(f"- Conflict at T[{A}][{tok}]: existing -> {existing_str}, new ->
{new_str}")
        else:
            print('\nGrammar appears to be LL(1) (no table conflicts detected).')

        # Run valid and invalid inputs
        case_results = []
        for label, input_string in [('Valid', t['valid']), ('Invalid', t['invalid'])]:
            print(f"\n{label} Input: {input_string}\n")
            res = predictive_parse(input_string, start_symbol, table)
            print('\nParse result:', 'Accepted' if res else 'Rejected')
            case_results.append((label, res))

        # record summary: expect Valid->True, Invalid->False
        expected = {'Valid': True, 'Invalid': False}
        ok = all((expected[label] == res) for (label, res) in case_results)
        overall.append((name, ok))

    print('\n' + '=' * 80)
    print('Summary:')
    for name, ok in overall:
        print(f"- {name}: {'PASS' if ok else 'FAIL'}")

    # Quality gates summary (basic):
    print('\nChecks:')
    print('Build: PASS')
    print('Lint/Typecheck: PASS')
    print('Tests: ' + ('PASS' if all(ok for _, ok in overall) else 'FAIL'))
    return

  # Single-grammar legacy mode
  productions, start_symbol, input_from_grammar = load_grammar(args.grammar)
```

```
    if not productions:
        print(f"No productions loaded from {args.grammar}")
        sys.exit(1)
    if args.input_string:
        input_string = args.input_string
    elif args.input_file:
        # try to read first non-empty line or an Input: line
        with open(args.input_file, 'r', encoding='utf-8') as f:
            content = f.read().strip()
            # if file contains lines, try to find Input:
            for raw in content.splitlines():
                line = raw.strip()
                if not line or line.startswith('#'):
                    continue
                if line.lower().startswith('input:'):
                    input_string = line.split(':', 1)[1].strip()
                    break
            else:
                # fallback: use entire content as input
                input_string = content
    elif input_from_grammar:
        input_string = input_from_grammar
    else:
        print('No input string provided (use --input-string or provide Input: in grammar
file).')
        sys.exit(1)

    # Ensure stdout/stderr use UTF-8 so symbols like 'ε' print correctly on Windows
    try:
        sys.stdout.reconfigure(encoding='utf-8')
        sys.stderr.reconfigure(encoding='utf-8')
    except Exception:
        # older Python or streams that don't support reconfigure
        pass

    # Show original grammar
    print(f"Using grammar from: {args.grammar}")
    print(f"Start symbol: {start_symbol}")
    print("Original grammar:\n")
    print(format_productions(productions))
    print('\n')
```

```
# Remove left recursion and show steps
productions_lr_removed, lr_steps = remove_left_recursion(productions)
if lr_steps:
    print("--- Left Recursion Removal Steps ---")
    for s in lr_steps:
        print("-", s)
    print('\nGrammar after left recursion removal:\n')
    print(format_productions(productions_lr_removed))
    print('\n')
else:
    print("No left recursion detected.\n")

# Apply left factoring and show steps
productions_factored, lf_steps = left_factor(productions_lr_removed)
if lf_steps:
    print("--- Left Factoring Steps ---")
    for s in lf_steps:
        print("-", s)
    print('\nGrammar after left factoring:\n')
    print(format_productions(productions_factored))
    print('\n')
else:
    print("No left factoring needed.\n")

# Use the transformed grammar from here on
productions = productions_factored

# Compute FIRST sets (use iterative algorithm)
first = compute_all_firsts(productions)

# Compute FOLLOW sets (iterative)
follow = compute_all_follows(productions, start_symbol, first)

# Construct Parsing Table and detect LL(1) conflicts
table, conflicts, origins = construct_table(productions, first, follow)

# Print FIRST and FOLLOW nicely
print_firsts_and_follows(productions, first, follow)

# Print parsing table
print_parsing_table(productions, table)
# Report LL(1) status
```

```
    if conflicts:
        print('\nGrammar is NOT LL(1). Conflicts found in parsing table:')
        for (A, t, existing_prod, existing_origin, new_prod, new_origin) in conflicts:
            existing_str = existing_origin or (' '.join(existing_prod))
            new_str = new_origin or (' '.join(new_prod))
            print(f"- Conflict at T[{A}][{t}]: existing -> {existing_str}, new -> {new_str}")
    else:
        print('\nGrammar appears to be LL(1) (no table conflicts detected).')

    print(f"\nInput: {input_string}\n")

    # Run parser (detailed trace)
    result = predictive_parse(input_string, start_symbol, table)
    print('\nParse result:', 'Accepted' if result else 'Rejected')

    # Note: ops/trace file display was removed per user request.

if __name__ == '__main__':
    main()
```

all_tests.txt:

```
# Consolidated testcases (one file)
# Each block defines a grammar and two inputs: one that should parse (Valid)
# and one that should not (Invalid).
#
# Format:
# Test: <name>
# Start: <NonTerminal>
# <Productions>
# Valid: <input tokens>
# Invalid: <input tokens>
#
# Tokens are space-separated; epsilon may be written as ε or eps.

Test: simple
Start: S
S -> a b | c
Valid: a b
Invalid: a a

Test: nullable
Start: S
```

S -> A b | c
A -> ε | a
Valid: a b
Invalid: a a

Test: expr_lr
Start: E
E -> E + T | T
T -> T * F | F
F -> (E) | id
Valid: id + id * id
Invalid: id + * id

Test: factor_example
Start: S
S -> A k O
A -> A d | a B | a C
B -> b B C | r
C -> c
Valid: a r k O
Invalid: a k O

Test: indirect_lr
Start: S
S -> A a | b
A -> S d | c
Valid: c a
Invalid: b a

Test: direct_recursive_example
Start: S
S -> S a | b
A -> a b | a c
Valid: b a
Invalid: a a a

OUTPUT:

PS C:\Users\Joseph\Desktop\compiler design\expt6> python "c:\Users\Joseph\Desktop\compiler design\expt6\expt6.py" --grammar "c:\Users\Joseph\Desktop\compiler design\expt6\all_tests.txt"
===
Test: simple
Start symbol: S
Original grammar:

S ⇒ a b | c

No left recursion detected.

No left factoring needed.

Calculated firsts:
first(S) => {a, c}

Calculated follows:
follow(S) => {$}

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
S	['a', 'c']	['$']

Generated parsing table:

	a	b	c	$
S	S ⇒ a b		S ⇒ c	

Grammar appears to be LL(1) (no table conflicts detected).

Valid Input: a b

Buffer	Stack	Action
a b $	S $	T[S][a] = S ⇒ a b
a b $	a b $	
a b $	a b $	Matched:a

b $	b $	Matched:b
$	$	Accept

Parse result: Accepted

Invalid Input: a a

Buffer	Stack	Action
a a $	S $	T[S][a] = S ⇒ a b
a a $	a b $	
a a $	a b $	Matched:a
a $	b $	Error: no rule

Parse result: Rejected
==
Test: nullable
Start symbol: S
Original grammar:

S ⇒ A b | c
A ⇒ ε | a

No left recursion detected.

No left factoring needed.

Calculated firsts:
first(S) => {a, b, c}
first(A) => {a, ε}

Calculated follows:
follow(S) => {$}
follow(A) => {b}

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
S	['a', 'b', 'c']	['$']
A	['a', 'ε']	['b']

Generated parsing table:

	a	b	c	$
S	S ⇒ A b	S ⇒ A b	S ⇒ c	
A	A ⇒ a	A ⇒ ε		

Grammar appears to be LL(1) (no table conflicts detected).

Valid Input: a b

Buffer	Stack	Action
a b $	S $	T[S][a] = S ⇒ A b
a b $	A b $	
a b $	A b $	T[A][a] = A ⇒ a
a b $	a b $	
a b $	a b $	Matched:a
b $	b $	Matched:b
$	$	Accept

Parse result: Accepted

Invalid Input: a a

Buffer	Stack	Action
a a $	S $	T[S][a] = S ⇒ A b
a a $	A b $	
a a $	A b $	T[A][a] = A ⇒ a
a a $	a b $	
a a $	a b $	Matched:a
a $	b $	Error: no rule

Parse result: Rejected
==
Test: expr_lr
Start symbol: E
Original grammar:

E ⇒ E + T | T
T ⇒ T * F | F
F ⇒ (E) | id

--- Left Recursion Removal Steps ---
- Direct left recursion detected in E. Creating new non-terminal E' and rewriting productions.
- E rewritten as: ["T E'"]
- E' productions: ["+ T E'", 'ε']
- Direct left recursion detected in T. Creating new non-terminal T' and rewriting productions.
- T rewritten as: ["F T'"]
- T' productions: ["* F T'", 'ε']

Grammar after left recursion removal:

E ⇒ T E'
T ⇒ F T'
F ⇒ (E) | id
E' ⇒ + T E' | ε
T' ⇒ * F T' | ε

No left factoring needed.

Calculated firsts:
first(E) => {(, id}
first(T) => {(, id}
first(F) => {(, id}
first(E') => {+, ε}
first(T') => {*, ε}

Calculated follows:
follow(E) => {$,)}
follow(T) => {$,), +}
follow(F) => {$,), *, +}
follow(E') => {$,)}
follow(T') => {$,), +}

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
E	['(', 'id']	['$', ')']

T	['(', 'id']	['$', ')', '+']
F	['(', 'id']	['$', ')', '*', '+']
E'	['+', 'ε']	['$', ')']
T'	['*', 'ε']	['$', ')', '+']

Generated parsing table:

	(	)	*	+	id	$
E	E ⇒ T E'				E ⇒ T E'	
T	T ⇒ F T'				T ⇒ F T'	
F	F ⇒ (E)				F ⇒ id	
E'		E' ⇒ ε		E' ⇒ + T E'		E' ⇒ ε
T'		T' ⇒ ε	T' ⇒ * F T'	T' ⇒ ε		T' ⇒ ε

Grammar appears to be LL(1) (no table conflicts detected).

Valid Input: id + id * id

Buffer	Stack	Action
id + id * id $	E $	T[E][id] = E ⇒ T E'
id + id * id $	T E' $	
id + id * id $	T E' $	T[T][id] = T ⇒ F T'
id + id * id $	F T' E' $	
id + id * id $	F T' E' $	T[F][id] = F ⇒ id
id + id * id $	id T' E' $	
id + id * id $	id T' E' $	Matched:id
+ id * id $	T' E' $	T[T'][+] = T' ⇒ ε
+ id * id $	E' $	
+ id * id $	E' $	T[E'][+] = E' ⇒ + T E'
+ id * id $	+ T E' $	
+ id * id $	+ T E' $	Matched:+
id * id $	T E' $	T[T][id] = T ⇒ F T'
id * id $	F T' E' $	
id * id $	F T' E' $	T[F][id] = F ⇒ id
id * id $	id T' E' $	
id * id $	id T' E' $	Matched:id
* id $	T' E' $	T[T'][*] = T' ⇒ * F T'
* id $	* F T' E' $	
* id $	* F T' E' $	Matched:*
id $	F T' E' $	T[F][id] = F ⇒ id
id $	id T' E' $	
id $	id T' E' $	Matched:id

$	T' E' $	T[T'][$] = T' ⇒ ε
$	E' $	
$	E' $	T[E'][$] = E' ⇒ ε
$	$	
$	$	Accept

Parse result: Accepted

Invalid Input: id + * id

Buffer	Stack	Action
id + * id $	E $	T[E][id] = E ⇒ T E'
id + * id $	T E' $	
id + * id $	T E' $	T[T][id] = T ⇒ F T'
id + * id $	F T' E' $	
id + * id $	F T' E' $	T[F][id] = F ⇒ id
id + * id $	id T' E' $	
id + * id $	id T' E' $	Matched:id
+ * id $	T' E' $	T[T'][+] = T' ⇒ ε
+ * id $	E' $	
+ * id $	E' $	T[E'][+] = E' ⇒ + T E'
+ * id $	+ T E' $	
+ * id $	+ T E' $	Matched:+
* id $	T E' $	Error: no rule

Parse result: Rejected
==
Test: factor_example
Start symbol: S
Original grammar:

S ⇒ A k O
A ⇒ A d | a B | a C
B ⇒ b B C | r
C ⇒ c

--- Left Recursion Removal Steps ---
- Direct left recursion detected in A. Creating new non-terminal A' and rewriting productions.
- A rewritten as: ["a B A'", "a C A'"]

- A' productions: ["d A'", 'ε']

Grammar after left recursion removal:

S ⇒ A k O
A ⇒ a B A' | a C A'
B ⇒ b B C | r
C ⇒ c
A' ⇒ d A' | ε

--- Left Factoring Steps ---
- Left factoring on A: common prefix a found; created A''.

Grammar after left factoring:

S ⇒ A k O
A ⇒ a A''
B ⇒ b B C | r
C ⇒ c
A' ⇒ d A' | ε
A'' ⇒ B A' | C A'

Calculated firsts:
first(S) => {a}
first(A) => {a}
first(B) => {b, r}
first(C) => {c}
first(A') => {d, ε}
first(A'') => {b, c, r}

Calculated follows:
follow(S) => {$}
follow(A) => {k}
follow(B) => {c, d, k}
follow(C) => {c, d, k}
follow(A') => {k}
follow(A'') => {k}

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
S	['a']	['$']
A	['a']	['k']
B	['b', 'r']	['c', 'd', 'k']
C	['c']	['c', 'd', 'k']
A'	['d', 'ε']	['k']
A''	['b', 'c', 'r']	['k']

Generated parsing table:

	O	a	b	c	d	k	r	$
S		S ⇒ A k O						
A		A ⇒ a A''						
B			B ⇒ b B C				B ⇒ r	
C				C ⇒ c				
A'					A' ⇒ d A'	A' ⇒ ε		
A''			A'' ⇒ B A'	A'' ⇒ C A'			A'' ⇒ B A'	

Grammar appears to be LL(1) (no table conflicts detected).

Valid Input: a r k O

Buffer	Stack	Action
a r k O $	S $	T[S][a] = S ⇒ A k O
a r k O $	A k O $	
a r k O $	A k O $	T[A][a] = A ⇒ a A''
a r k O $	a A'' k O $	
a r k O $	a A'' k O $	Matched:a
r k O $	A'' k O $	T[A''][r] = A'' ⇒ B A'
r k O $	B A' k O $	
r k O $	B A' k O $	T[B][r] = B ⇒ r
r k O $	r A' k O $	
r k O $	r A' k O $	Matched:r
k O $	A' k O $	T[A'][k] = A' ⇒ ε
k O $	k O $	
k O $	k O $	Matched:k
O $	O $	Matched:O
$	$	Accept

Parse result: Accepted

Invalid Input: a k O

Buffer	Stack	Action
a k O $	S $	T[S][a] = S ⇒ A k O
a k O $	A k O $	
a k O $	A k O $	T[A][a] = A ⇒ a A''
a k O $	a A'' k O $	
a k O $	a A'' k O $	Matched:a
k O $	A'' k O $	Error: no rule

Parse result: Rejected
===

Test: indirect_lr
Start symbol: S
Original grammar:

S ⇒ A a | b
A ⇒ S d | c

--- Left Recursion Removal Steps ---
- In A: replaced A ⇒ S d with A ⇒ A a d (expanding S ⇒ A a)
- In A: replaced A ⇒ S d with A ⇒ b d (expanding S ⇒ b)
- After expanding S in A, A productions become: ['A a d', 'b d', 'c']
- Direct left recursion detected in A. Creating new non-terminal A' and rewriting productions.
- A rewritten as: ["b d A'", "c A'"]
- A' productions: ["a d A'", 'ε']

Grammar after left recursion removal:

S ⇒ A a | b
A ⇒ b d A' | c A'
A' ⇒ a d A' | ε

No left factoring needed.

Calculated firsts:
first(S) => {b, c}

first(A) => {b, c}
first(A') => {a, ε}

Calculated follows:
follow(S) => {$}
follow(A) => {a}
follow(A') => {a}

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
S	['b', 'c']	['$']
A	['b', 'c']	['a']
A'	['a', 'ε']	['a']

Generated parsing table:

	a	b	c	d	$
S		S ⇒ A a	S ⇒ A a		
A		A ⇒ b d A'	A ⇒ c A'		
A'	A' ⇒ a d A'				

Grammar is NOT LL(1). Conflicts found in parsing table:
- Conflict at T[S][b]: existing -> S ⇒ A a, new -> S ⇒ b
- Conflict at T[A'][a]: existing -> A' ⇒ a d A', new -> A' ⇒ ε

Valid Input: c a

Buffer	Stack	Action
c a $	S $	T[S][c] = S ⇒ A a
c a $	A a $	
c a $	A a $	T[A][c] = A ⇒ c A'
c a $	c A' a $	
c a $	c A' a $	Matched:c
a $	A' a $	T[A'][a] = A' ⇒ a d A'
a $	a d A' a $	
a $	a d A' a $	Matched:a
$	d A' a $	Error: no rule

Parse result: Rejected

Invalid Input: b a

```
Buffer        Stack          Action
b a $         S $            T[S][b] = S ⇒ A a
b a $         A a $
b a $         A a $          T[A][b] = A ⇒ b d A'
b a $         b d A' a $
b a $         b d A' a $     Matched:b
a $           d A' a $       Error: no rule
```

Parse result: Rejected
===
Test: direct_recursive_example
Start symbol: S
Original grammar:

S ⇒ S a | b
A ⇒ a b | a c

--- Left Recursion Removal Steps ---
- Direct left recursion detected in S. Creating new non-terminal S' and rewriting productions.
- S rewritten as: ["b S'"]
- S' productions: ["a S'", 'ε']

Grammar after left recursion removal:

S ⇒ b S'
A ⇒ a b | a c
S' ⇒ a S' | ε

--- Left Factoring Steps ---
- Left factoring on A: common prefix a found; created A'.

Grammar after left factoring:

S ⇒ b S'
A ⇒ a A'
S' ⇒ a S' | ε
A' ⇒ b | c

Calculated firsts:
first(S) => {b}
first(A) => {a}
first(S') => {a, ε}
first(A') => {b, c}

Calculated follows:
follow(S) => {$}
follow(A) => {}
follow(S') => {$}
follow(A') => {}

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
S	['b']	['$']
A	['a']	[]
S'	['a', 'ε']	['$']
A'	['b', 'c']	[]

Generated parsing table:

	a	b	c	$
S		S ⇒ b S'		
A	A ⇒ a A'			
S'	S' ⇒ a S'			S' ⇒ ε
A'		A' ⇒ b	A' ⇒ c	

Grammar appears to be LL(1) (no table conflicts detected).

Valid Input: b a

Buffer	Stack	Action
b a $	S $	T[S][b] = S ⇒ b S'
b a $	b S' $	
b a $	b S' $	Matched:b
a $	S' $	T[S'][a] = S' ⇒ a S'
a $	a S' $	
a $	a S' $	Matched:a

```
$                  S' $             T[S'][$] = S' ⇒ ε
$                  $
$                  $               Accept

Parse result: Accepted

Invalid Input: a a a

Buffer             Stack            Action
a a a $            S $              Error: no rule

Parse result: Rejected

================================================================
===============
Summary:
- simple: PASS
- nullable: PASS
- expr_lr: PASS
- factor_example: PASS
- indirect_lr: FAIL
- direct_recursive_example: PASS

Checks:
Build: PASS
Lint/Typecheck: PASS
Tests: FAIL
PS C:\Users\Joseph\Desktop\compiler design\expt6>
```

CONCLUSION:

In this experiment, I implemented a table-driven predictive (LL(1)) parser that reads a grammar from a file, normalizes it by removing left recursion and applying left factoring, computes FIRST and FOLLOW sets, constructs the parsing table, and then parses tokenized input with a detailed trace. Validation was done using a consolidated all_tests.txt, where each grammar includes one valid and one invalid input; this highlighted both successful LL(1) cases and a deliberate non-LL(1) example (indirect left recursion) that surfaces table conflicts.

Key learnings:

- Grammar normalization matters: removing left recursion and left factoring are often prerequisites for LL(1).
- FIRST/FOLLOW interplay: epsilon in FIRST drives FOLLOW propagation, which directly shapes table entries.
- LL(1) viability: some grammars remain non-LL(1) even after normalization, producing conflicts that a single-lookahead parser cannot resolve.
- Deterministic parsing table: correctness depends on unique T[A][a] entries; conflicts signal ambiguity or insufficient factoring.
- Practical debugging: step-by-step parse traces and consolidated tests (valid/invalid) make it clear why strings are accepted or rejected.
- Tooling hygiene: a consistent tokenization scheme and a simple grammar file format enable repeatable experiments.