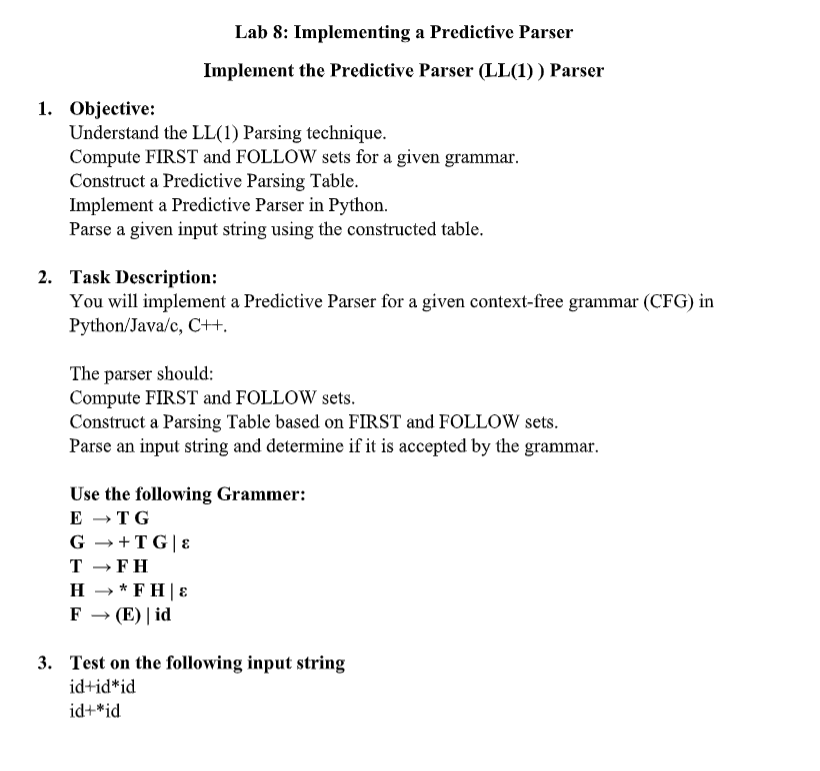


|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | COMPILER DESIGN | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | SUBMITTED BY: |  | | | |
|  | | | | SAMAR MITTALMIS: 112315159FACULTY:DR MAYANK LOVANSHI& DR KAPTAN SINGH |  | | | |
|  | | |  | | |  | | |



## PYTHON CODE :

rule\_set = [

    ("E", ["T","G"]),

    ("G", ["+","T","G"]),

    ("G", ["#"]),

    ("T", ["F","H"]),

    ("H", [ "\*", "F",'H']),

    ("H", [ "#"]),

    ("F", ["(", "E", ")"]),

    ("F", ["id"])

]

tokens = {"id", "+", "\*", "(", ")", "$"}

symbols = {"E",'G','H', "T", "F"}

transformed\_rules = []

first\_set, follow\_set, parse\_table = {}, {}, {}

def remove\_left\_recursion():

    global symbols, transformed\_rules

    new\_rules, new\_symbols = [], set(symbols)

    for symbol in symbols:

        direct\_recur, other\_prod = [], []

        for rule in rule\_set:

            if rule[0] == symbol:

                (direct\_recur if rule[1] and rule[1][0] == symbol else other\_prod).append(

                    rule[1][1:] if rule[1] and rule[1][0] == symbol else rule[1]

                )

        if not direct\_recur:

            [new\_rules.append((symbol, prod)) for prod in other\_prod]

            continue

        new\_symbol = symbol + "'"

        new\_symbols.add(new\_symbol)

        [new\_rules.append((symbol, prod + [new\_symbol])) for prod in other\_prod]

        [new\_rules.append((new\_symbol, recur + [new\_symbol])) for recur in direct\_recur]

        new\_rules.append((new\_symbol, ["#"]))

    transformed\_rules.extend(new\_rules)

    symbols = new\_symbols

    print("Grammar after removing left recursion:")

    for rule in transformed\_rules:

        print(f"{rule[0]} → {' '.join(rule[1])}")

    print()

def remove\_left\_factoring():

    global symbols, transformed\_rules

    new\_rules = []

    for symbol in symbols:

        prods = [rule[1] for rule in transformed\_rules if rule[0] == symbol]

        i = 0

        while i < len(prods):

            for j in range(i + 1, len(prods)):

                if prods[i] and prods[j] and prods[i][0] == prods[j][0]:

                    new\_symbol = f"{symbol}\_{len(new\_rules)}"

                    symbols.add(new\_symbol)

                    new\_rules.append((symbol, [prods[i][0], new\_symbol]))

                    new\_rules.append((new\_symbol, prods[i][1:] or ["#"]))

                    new\_rules.append((new\_symbol, prods[j][1:] or ["#"]))

                    prods.pop(j)

                    prods.pop(i)

                    i -= 1

                    break

            i += 1

        [new\_rules.append((symbol, prod)) for prod in prods]

    transformed\_rules.clear()

    transformed\_rules.extend(new\_rules)

    print("Grammar after removing left factoring:")

    for rule in transformed\_rules:

        print(f"{rule[0]} → {' '.join(rule[1])}")

    print()

def compute\_first\_set():

    for token in tokens | symbols:

        first\_set[token] = set()

    for token in tokens:

        if token != "$":

            first\_set[token].add(token)

    changed = True

    while changed:

        changed = False

        for left\_side, right\_side in transformed\_rules:

            original\_size = len(first\_set[left\_side])

            if right\_side[0] == "#":

                first\_set[left\_side].add("#")

            else:

                nullable = True

                for symbol in right\_side:

                    if not nullable:

                        break

                    first\_set[left\_side].update(s for s in first\_set[symbol] if s != "#")

                    nullable = "#" in first\_set[symbol]

                    if nullable and symbol == right\_side[-1]:

                        first\_set[left\_side].add("#")

            changed |= len(first\_set[left\_side]) > original\_size

def compute\_follow\_set():

    for symbol in symbols:

        follow\_set.setdefault(symbol, set())

    follow\_set["E"].add("$")

    changed = True

    while changed:

        changed = False

        for left\_side, right\_side in transformed\_rules:

            for index, symbol in enumerate(right\_side):

                if symbol in symbols:

                    original\_size = len(follow\_set[symbol])

                    nullable = True

                    for next\_symbol\_index in range(index + 1, len(right\_side)):

                        if not nullable:

                            break

                        follow\_set[symbol].update(first\_set[right\_side[next\_symbol\_index]] - {"#"})

                        nullable = "#" in first\_set[right\_side[next\_symbol\_index]]

                    if nullable or index == len(right\_side) - 1:

                        follow\_set[symbol].update(follow\_set[left\_side])

                    changed |= len(follow\_set[symbol]) > original\_size

def construct\_parse\_table():

    for left\_side, right\_side in transformed\_rules:

        if right\_side == ["#"]:

            first\_production = {"#"}

        else:

            first\_production = set().union(

                \*(first\_set[right\_side[i]] - {"#"} for i in range(len(right\_side)) if all("#" in first\_set[right\_side[j]] for j in range(i)))

            )

            if all("#" in first\_set[symbol] for symbol in right\_side if symbol != "#"):

                first\_production.add("#")

        for token in first\_production - {"#"}:

            parse\_table[(left\_side, token)] = right\_side

        if right\_side == ["#"]:

            for token in follow\_set[left\_side]:

                parse\_table[(left\_side, token)] = right\_side

def print\_sets():

    print(f"{"SYMBOLS":<10}{"FIRST":<10}{"FOLLOW":<10}")

    for symbol in symbols:

        print(f"{symbol:<10}{','.join(sorted(first\_set[symbol])):<10}{','.join(sorted(follow\_set[symbol])):<10}")

    print()

def print\_parse\_table():

    print("Predictive Parsing Table:\n")

    header = ["NT/T"] + list(tokens)

    row\_format = "{:<12}" \* len(header)

    print(row\_format.format(\*header))

    print("-" \* (12 \* len(header)))

    for symbol in symbols:

        row = [symbol]

        for token in tokens:

            row.append(f"{symbol} → {' '.join(parse\_table[symbol, token])}" if (symbol, token) in parse\_table else "")

        print(row\_format.format(\*row))

        print("-" \* (12 \* len(header)))

def parse\_input\_string(input\_string):

    input\_string += "$"

    parse\_stack = ["$", "E"]

    input\_index = 0

    print(f"Parsing: {input\_string[:-1]}\n")

    print(f"{'Stack':<30}{'Input':<20}{'Action'}")

    print("=" \* 70)

    while parse\_stack:

        top\_of\_stack, current\_input\_symbol = parse\_stack[-1], "id" if input\_string[input\_index:].startswith("id") else input\_string[input\_index]

        print(f"{' '.join(parse\_stack):<30}{input\_string[input\_index:]:<20}", end="")

        if top\_of\_stack in tokens and top\_of\_stack == current\_input\_symbol:

            parse\_stack.pop()

            input\_index += 2 if top\_of\_stack == "id" else 1

            print(f"Match {top\_of\_stack}")

        elif top\_of\_stack == "#":

            parse\_stack.pop()

            print("Pop #")

        elif top\_of\_stack in symbols and (top\_of\_stack, current\_input\_symbol) in parse\_table:

            production = parse\_table[(top\_of\_stack, current\_input\_symbol)]

            parse\_stack.pop()

            if production[0] != "#":

                for symbol in reversed(production):

                    parse\_stack.append(symbol)

            print(f"{top\_of\_stack} → {' '.join(production)}")

        else:

            print(f"Error: No entry for ({top\_of\_stack}, {current\_input\_symbol})")

            return False

    return input\_index == len(input\_string)

print("Original Grammar:")

for rule in rule\_set:

    print(f"{rule[0]} → {' '.join(rule[1])}")

print()

remove\_left\_recursion()

remove\_left\_factoring()

compute\_first\_set()

compute\_follow\_set()

construct\_parse\_table()

print\_sets()

print\_parse\_table()

for test\_string in ["id+id\*id", "id+\*id"]:

    print(f"\nInput '{test\_string}' is {'accepted' if parse\_input\_string(test\_string) else 'not accepted'}")

    print("-" \* 50)

## OUTPUT :



