COMPILER DESIGN

SUBMITTED BY:

SAMAR MITTAL
MIS: 112315159

FACULTY:
DR MAYANK LOVANSHI
& DR KAPTAN SINGH

Lab 8: Implementing a Predictive Parser

Implement the Predictive Parser (LL(1)) Parser

1. **Objective:**
   Understand the LL(1) Parsing technique.
   Compute FIRST and FOLLOW sets for a given grammar.
   Construct a Predictive Parsing Table.
   Implement a Predictive Parser in Python.
   Parse a given input string using the constructed table.

2. **Task Description:**
   You will implement a Predictive Parser for a given context-free grammar (CFG) in Python/Java/c, C++.

   The parser should:
   Compute FIRST and FOLLOW sets.
   Construct a Parsing Table based on FIRST and FOLLOW sets.
   Parse an input string and determine if it is accepted by the grammar.

   Use the following Grammer:
   E → T G
   G → + T G | ε
   T → F H
   H → * F H | ε
   F → (E) | id

3. **Test on the following input string**
   id+id*id
   id+*id

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 :

```
PS C:\Users\Samar Mittal> python  'c:\Users\Samar Mittal\Desktop\Compiler LAb\lab9\temp.py'
Original Grammar:
E → T G
G → + T G
G → #
T → F H
H → * F H
H → #
F → ( E )
F → id

Grammar after removing left recursion:
F → ( E )
F → id
G → + T G
G → #
H → * F H
H → #
E → T G
T → F H

Grammar after removing left factoring:
F → ( E )
F → id
G → + T G
G → #
H → * F H
H → #
E → T G
T → F H
 SYMBOLS   FIRST     FOLLOW
 F         (,id      $,),*,+
 G         #,+       $,)
 H         #,*       $,),+
 E         (,id      $,)
 T         (,id      $,),+

 Predictive Parsing Table:

 NT/T       +          id          *          )          $          (
 ------------------------------------------------------------------------------
 F                     F → id                                   F → ( E )
 ------------------------------------------------------------------------------
 G          G → + T G                     G → #      G → #
 ------------------------------------------------------------------------------
 H          H → #                 H → * F H  H → #      H → #
 ------------------------------------------------------------------------------
 E                     E → T G                                  E → T G
 ------------------------------------------------------------------------------
 T                     T → F H                                  T → F H
 ------------------------------------------------------------------------------
 Parsing: id+id*id
```

```
Parsing: id+id*id

Stack                           Input               Action
=================================================================
$ E                             id+id*id$           E → T G
$ G T                           id+id*id$           T → F H
$ G H F                         id+id*id$           F → id
$ G H id                        id+id*id$           Match id
$ G H                           +id*id$             H → #
$ G                             +id*id$             G → + T G
$ G T +                         +id*id$             Match +
$ G T                           id*id$              T → F H
$ G H F                         id*id$              F → id
$ G H id                        id*id$              Match id
$ G H                           *id$                H → * F H
$ G H F *                       *id$                Match *
$ G H F                         id$                 F → id
$ G H id                        id$                 Match id
$ G H                           $                   H → #
$ G                             $                   G → #
$                               $                   Match $

Input 'id+id*id' is accepted
-------------------------------------------------
Parsing: id+*id

Stack                           Input               Action
=================================================================
$ E                             id+*id$             E → T G
$ G T                           id+*id$             T → F H
$ G H F                         id+*id$             F → id
$ G H id                        id+*id$             Match id
$ G H                           +*id$               H → #
$ G                             +*id$               G → + T G
$ G T +                         +*id$               Match +
$ G T                           *id$                Error: No entry for (T, *)

Input 'id+*id' is not accepted
-------------------------------------------------
PS C:\Users\Samar Mittal>
```