PROJECT:

LL (1) PARSING

|  |  |
| --- | --- |
| **NAME** | **STUDENT-ID** |
| SAHIBA SHAHAB | 12287 |
| HIBA BINT IRFAN | 13146 |

# **INTRODUCTION**

The Context-Free Grammar Parser Generator project aims to provide a comprehensive tool for working with CFGs. It offers functionalities to compute the First and Follow sets for non-terminals, generate parsing tables, and implement parsing algorithms. With this tool, users can define grammars, analyze their properties, and perform parsing operations efficiently.

# **OBJECTIVE**

The primary objective of the Context-Free Grammar Parser Generator project is to develop a versatile and user-friendly tool that enables efficient manipulation and parsing of context-free grammars. The project aims to address the challenges associated with parsing algorithms and provide a comprehensive solution for developers, researchers, and students working in formal language theory and parsing techniques.

# **SCOPE**

The project's scope encompasses the development of a parser generator tool with the following key features:

* Removal of Left Recursion: The project includes an algorithm to eliminate left recursion from the grammar rules, which is a prerequisite for LL(1) parsing.
* Left Factoring: Another feature is left factoring, which helps in resolving ambiguities in the grammar rules.
* First and Follow Sets Computation: The project calculates the First and Follow sets for each non-terminal symbol in the grammar.
* Parsing Table Generation: It generates a parsing table based on the computed First and Follow sets, which is crucial for LL(1) parsing.
* Input String Validation: The parser validates input strings against the grammar rules using the parsing table and stack-buffer mechanism.
* GUI Interface: The project provides a user-friendly graphical interface using Tkinter library for inputting grammar rules and sample input strings, and displaying validation results.

Usage:

* The user can input grammar rules, non-terminal and terminal symbols, and sample input strings using the graphical interface.
* After inputting the required data, the user can click the "Submit" button to trigger the parsing process.
* The validation result (whether the input string is valid or not) is displayed in the result panel of the interface.

# **TECHNOLOGIES**

Python: Core programming language for implementing the parser generator and algorithms.

# **FEATURES**

1. Grammar Input
2. Recursive
3. Left factoring
4. First Set Calculation
5. Follow Set Calculation
6. Parsing Table Generation
7. Stack table
8. String input
9. Parsing Algorithms

**CODE**

import tkinter as tk

def removeLeftRecursion(rulesDiction):

    store = {}

    for lhs in rulesDiction:

        alphaRules = []

        betaRules = []

        allrhs = rulesDiction[lhs]

        for subrhs in allrhs:

            if subrhs[0] == lhs:

                alphaRules.append(subrhs[1:])

            else:

                betaRules.append(subrhs)

        if len(alphaRules) != 0:

            lhs\_ = lhs + "'"

            while lhs\_ in rulesDiction.keys() or lhs\_ in store.keys():

                lhs\_ += "'"

            for b in range(0, len(betaRules)):

                betaRules[b].append(lhs\_)

            rulesDiction[lhs] = betaRules

            for a in range(0, len(alphaRules)):

                alphaRules[a].append(lhs\_)

            alphaRules.append(['#'])

            store[lhs\_] = alphaRules

    for left in store:

        rulesDiction[left] = store[left]

    return rulesDiction

def LeftFactoring(rulesDiction):

    newDict = {}

    for lhs in rulesDiction:

        allrhs = rulesDiction[lhs]

        temp = dict()

        for subrhs in allrhs:

            if subrhs[0] not in list(temp.keys()):

                temp[subrhs[0]] = [subrhs]

            else:

                temp[subrhs[0]].append(subrhs)

        new\_rule = []

        tempo\_dict = {}

        for term\_key in temp:

            allStartingWithTermKey = temp[term\_key]

            if len(allStartingWithTermKey) > 1:

                lhs\_ = lhs + "'"

                while lhs\_ in rulesDiction.keys() or lhs\_ in tempo\_dict.keys():

                    lhs\_ += "'"

                new\_rule.append([term\_key, lhs\_])

                ex\_rules = []

                for g in temp[term\_key]:

                    ex\_rules.append(g[1:])

                tempo\_dict[lhs\_] = ex\_rules

            else:

                new\_rule.append(allStartingWithTermKey[0])

        newDict[lhs] = new\_rule

        for key in tempo\_dict:

            newDict[key] = tempo\_dict[key]

    return newDict

def first(rule, rules, diction, firsts, term\_userdef):

    if len(rule) != 0 and (rule is not None):

        if rule[0] in term\_userdef:

            return rule[0]

        elif rule[0] == '#':

            return '#'

    if len(rule) != 0:

        if rule[0] in list(diction.keys()):

            fres = []

            rhs\_rules = diction[rule[0]]

            for itr in rhs\_rules:

                indivRes = first(itr, rules, diction, firsts, term\_userdef)

                if isinstance(indivRes, list):

                    for i in indivRes:

                        fres.append(i)

                else:

                    fres.append(indivRes)

            if '#' not in fres:

                return fres

            else:

                newList = []

                fres.remove('#')

                if len(rule) > 1:

                    ansNew = first(rule[1:], rules, diction, firsts, term\_userdef)

                    if ansNew is not None:

                        if isinstance(ansNew, list):

                            newList = fres + ansNew

                        else:

                            newList = fres + [ansNew]

                    else:

                        newList = fres

                    return newList

                fres.append('#')

                return fres

def follow(nt, start\_symbol, rules, nonterm\_userdef, term\_userdef, diction, firsts, follows):

    solset = set()

    if nt == start\_symbol:

        solset.add('$')

    for curNT in diction:

        rhs = diction[curNT]

        for subrule in rhs:

            if nt in subrule:

                res = None

                while nt in subrule:

                    index\_nt = subrule.index(nt)

                    subrule = subrule[index\_nt + 1:]

                    if len(subrule) != 0:

                        res = first(subrule, rules, diction, firsts,term\_userdef)

                        if res is None:

                            continue

                        if '#' in res:

                            newList = []

                            res.remove('#')

                            ansNew = follow(curNT, start\_symbol, rules, nonterm\_userdef, term\_userdef, diction, firsts, follows)

                            if ansNew is not None:

                                if isinstance(ansNew, list):

                                    newList = res + ansNew

                                else:

                                    newList = res + [ansNew]

                            else:

                                newList = res

                            res = newList

                    else:

                        if nt != curNT:

                            res = follow(curNT, start\_symbol, rules, nonterm\_userdef, term\_userdef, diction, firsts, follows)

                    if res is not None:

                        if isinstance(res, list):

                            for g in res:

                                solset.add(g)

                        else:

                            solset.add(res)

    return list(solset)

def computeAllFirsts(rules, nonterm\_userdef, term\_userdef, diction, firsts):

   for rule in rules:

    k = rule.split("->")

    k[0] = k[0].strip()

    k[1] = k[1].strip()

    rhs = k[1]

    multirhs = rhs.split('|')

    for i in range(len(multirhs)):

        multirhs[i] = multirhs[i].strip()

        multirhs[i] = multirhs[i].split()

    diction[k[0]] = multirhs

    print(f"\nRules: \n")

    for y in diction:

        print(f"{y}->{diction[y]}")

    print(f"\nAfter elimination of left recursion:\n")

    diction = removeLeftRecursion(diction)

    for y in diction:

        print(f"{y}->{diction[y]}")

    print("\nAfter left factoring:\n")

    diction = LeftFactoring(diction)

    for y in diction:

        print(f"{y}->{diction[y]}")

    for y in list(diction.keys()):

        t = set()

        for sub in diction.get(y):

            res = first(sub, rules, diction, firsts, term\_userdef)

            if res is not None:

                if isinstance(res, list):

                    for u in res:

                        t.add(u)

                else:

                    t.add(res)

        firsts[y] = t

    print("\nCalculated firsts: ")

    key\_list = list(firsts.keys())

    index = 0

    for gg in firsts:

        print(f"first({key\_list[index]}) "

            f"=> {firsts.get(gg)}")

        index += 1

def computeAllFollows(start\_symbol, rules, nonterm\_userdef, term\_userdef, diction, firsts, follows):

    for NT in diction:

        solset = set()

        sol = follow(NT, start\_symbol, rules, nonterm\_userdef, term\_userdef, diction, firsts, follows)

        if sol is not None:

            for g in sol:

                solset.add(g)

        follows[NT] = solset

    print("\nCalculated follows: ")

    key\_list = list(follows.keys())

    index = 0

    for gg in follows:

        print(f"follow({key\_list[index]})"

            f" => {follows[gg]}")

        index += 1

def createParseTable(diction, firsts, rules, follows, term\_userdef):

    import copy

    print("\nFirsts and Follow Result table\n")

    mx\_len\_first = 0

    mx\_len\_fol = 0

    for u in diction.keys():

        k1 = len(str(firsts[u]))

        k2 = len(str(follows[u]))

        if k1 > mx\_len\_first:

            mx\_len\_first = k1

        if k2 > mx\_len\_fol:

            mx\_len\_fol = k2

    print(f"{{:<{10}}} "

          f"{{:<{mx\_len\_first + 5}}} "

          f"{{:<{mx\_len\_fol + 5}}}"

          .format("Non-T", "FIRST", "FOLLOW"))

    for u in diction:

        print(f"{{:<{10}}} "

              f"{{:<{mx\_len\_first + 5}}} "

              f"{{:<{mx\_len\_fol + 5}}}"

              .format(u, str(firsts[u]), str(follows[u])))

    ntlist = list(diction.keys())

    terminals = copy.deepcopy(term\_userdef)

    terminals.append('$')

    mat = []

    for x in diction:

        row = []

        for y in terminals:

            row.append('')

        mat.append(row)

    grammar\_is\_LL = True

    for lhs in diction:

        rhs = diction[lhs]

        for y in rhs:

            res = first(y, rules, diction, firsts, term\_userdef)

            if res is None:

                continue

            if '#' in res:

                if isinstance(res, str):

                    firstFollow = []

                    fol\_op = follows[lhs]

                    if fol\_op is str:

                        firstFollow.append(fol\_op)

                    else:

                        for u in fol\_op:

                            firstFollow.append(u)

                    res = firstFollow

                else:

                    res.remove('#')

                    res = list(res) + list(follows[lhs])

            ttemp = []

            if isinstance(res, str):

                ttemp.append(res)

                res = copy.deepcopy(ttemp)

            for c in res:

                print("Current c:", c)

                if c is not None:

                    xnt = ntlist.index(lhs)

                    yt = terminals.index(c)

                    if mat[xnt][yt] == '':

                        mat[xnt][yt] = mat[xnt][yt] + f"{lhs}->{' '.join(y)}"

                    else:

                        if f"{lhs}->{y}" in mat[xnt][yt]:

                            continue

                        else:

                            grammar\_is\_LL = False

                            mat[xnt][yt] = mat[xnt][yt] + f",{lhs}->{' '.join(y)}"

    print("\nGenerated parsing table:\n")

    frmt = "{:>12}" \* len(terminals)

    print(frmt.format(\*terminals))

    j = 0

    for y in mat:

        frmt1 = "{:>12}" \* len(y)

        print(f"{ntlist[j]} {frmt1.format(\*y)}")

        j += 1

    return (mat, grammar\_is\_LL, terminals)

def validateStringUsingStackBuffer(parsing\_table, grammarll1, table\_term\_list, input\_string, term\_userdef, start\_symbol, diction, rules):

    print(f"\nValidate String => {input\_string}\n")

    if grammarll1 == False:

        return f"\nInput String = " \

            f"\"{input\_string}\"\n" \

            f"Grammar is not LL(1)"

    stack = [start\_symbol, '$']

    buffer = []

    input\_string = input\_string.split()

    input\_string.reverse()

    buffer = ['$'] + input\_string

    print("{:>20} {:>20} {:>20}".

        format("Buffer", "Stack","Action"))

    while True:

        if stack == ['$'] and buffer == ['$']:

            print("{:>20} {:>20} {:>20}"

                .format(' '.join(buffer),

                        ' '.join(stack),

                        "Valid"))

            return "\nValid String!"

        elif stack[0] not in term\_userdef:

            x = list(diction.keys()).index(stack[0])

            y = table\_term\_list.index(buffer[-1])

            if parsing\_table[x][y] != '':

                entry = parsing\_table[x][y]

                print("{:>20} {:>20} {:>25}".

                    format(' '.join(buffer),

                            ' '.join(stack),

                            f"T[{stack[0]}][{buffer[-1]}] = {entry}"))

                lhs\_rhs = entry.split("->")

                lhs\_rhs[1] = lhs\_rhs[1].replace('#', '').strip()

                entryrhs = lhs\_rhs[1].split()

                stack = entryrhs + stack[1:]

            else:

                return f"\nInvalid String! No rule at " \

                    f"Table[{stack[0]}][{buffer[-1]}]."

        else:

            if stack[0] == buffer[-1]:

                print("{:>20} {:>20} {:>20}"

                    .format(' '.join(buffer),

                            ' '.join(stack),

                            f"Matched:{stack[0]}"))

                buffer = buffer[:-1]

                stack = stack[1:]

            else:

                return "\nInvalid String! " \

                    "Unmatched terminal symbols"

def parse\_input():

    rules\_input = rules\_text.get("1.0", tk.END).strip().split("\n")

    nonterm\_userdef\_input = nonterm\_text.get().strip().split(",")

    term\_userdef\_input = term\_text.get().strip().split(",")

    sample\_input\_string\_input = sample\_input\_text.get().strip()

    diction = {}

    firsts = {}

    follows = {}

    computeAllFirsts(rules\_input, nonterm\_userdef\_input, term\_userdef\_input, diction, firsts)

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

    computeAllFollows(start\_symbol, rules\_input, nonterm\_userdef\_input, term\_userdef\_input, diction, firsts, follows)

    (parsing\_table, result, tabTerm) = createParseTable(diction, firsts, rules\_input, follows, term\_userdef\_input)

    if sample\_input\_string\_input != "":

        validity = validateStringUsingStackBuffer(parsing\_table, result, tabTerm, sample\_input\_string\_input, term\_userdef\_input, start\_symbol, diction, rules\_input)

        result\_text.delete("1.0", tk.END)

        result\_text.insert(tk.END, validity)

root = tk.Tk()

root.title("LL Parser")

rules\_label = tk.Label(root, text="Rules:")

rules\_label.grid(row=0, column=0, sticky="w")

rules\_text = tk.Text(root, height=6, width=50)

rules\_text.grid(row=0, column=1)

nonterm\_label = tk.Label(root, text="Nonterminal User Definition:")

nonterm\_label.grid(row=1, column=0, sticky="w")

nonterm\_text = tk.Entry(root)

nonterm\_text.grid(row=1, column=1)

term\_label = tk.Label(root, text="Terminal User Definition:")

term\_label.grid(row=2, column=0, sticky="w")

term\_text = tk.Entry(root)

term\_text.grid(row=2, column=1)

sample\_input\_label = tk.Label(root, text="Sample Input String:")

sample\_input\_label.grid(row=3, column=0, sticky="w")

sample\_input\_text = tk.Entry(root)

sample\_input\_text.grid(row=3, column=1)

submit\_button = tk.Button(root, text="Submit", command=parse\_input)

submit\_button.grid(row=4, column=1)

result\_label = tk.Label(root, text="Result:")

result\_label.grid(row=5, column=0, sticky="w")

result\_text = tk.Text(root, height=6, width=50)

result\_text.grid(row=5, column=1)

root.mainloop()