PUNE INSTITUTE OF COMPUTER TECHNOLOGY,
DHANKAWADI, PUNE-43.

A Mini-Project Report
On
LL (1) Parsing Table Generation

SUBMITTED BY

Arpit Bhasin (4403)
Neel Bafna (4404)
Jayprakash Chawla (4411)

CLASS: BE-4

Under The Guidance of
Prof. B. D. Zope

COMPUTER ENGINEERING DEPARTMENT
Academic Year: 2019-20

PUNE INSTITUTE OF COMPUTER TECHNOLOGY,
DHANKAWADI PUNE-43.

CERTIFICATE

This is to certify that Mr. Jayprakash Chawla(4411) of B.E. (Computer Engineering Department) Batch 2019-2020, has sat-isfactorily completed a project report on “LL (1) Parsing Table Generation” towards the partial fulfillment of the fourth year Computer Engineering Semester II of Pune University.

Prof. B. D. Zope	Prof. M. S. Takalikar
Project Guide	Head of Department,
	Computer Engineering

Date: _____

Place: _____

Abstract:

In any programming language, parsing is performed after scanning each and every lexical unit of the program. The Lexical unit of Programming Language domain is any keyword, identifier, constant, operator, etc. Once all the units have been identified parsing is performed. Parser checks the syntax of each and statement of PL. If the syntax is found correct, the parser generates parse trees. LL (1) parsing is a top down parsing technique. LL (1) parser constructs the table and based on the entry in the table it chooses the production on the right hand side of the non-terminal.

KEYWORDS:

Lexical Unit, Parser, Syntax, Parse Tree, Top-Down Parsing.

LL (1) Parsing Table Generation

Contents

List of Figures

1 Introduction

1.1 Introduction

Parsing is performed during the syntax analysis phase. Syntax analysis is carried out by parser to check the validity of the source statement. It checks whether the syntax of the statement is correct or not and generates a parse tree in case of a valid statement and it causes an error if the statement is found invalid.

Parsing can be performed in two ways:

Top Down Parsing

Bottom Up Parsing

1.2 Top Down Parsing

Top-Down parsing is a strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rules of a formal grammar. Top-down parsing analyzes unknown data by hypothesizing general parse tree structures and then considering whether the known fundamental structures are compatible with the hypothesis. It occurs in the analysis of both natural languages and computer languages. Top-down parsing can be viewed as an attempt to find leftmost derivations of an input-stream by searching for parse-trees using a top-down expansion of the given formal grammar rules.

1.3 LL (1) Parser

LL (1) parsing is a predictive parsing technique. In LL (1) parsing the first L means scanning of string takes place from left to right and second L means that during parsing we choose leftmost non-terminal for derivation. The (1) in LL (1) means we have one look-ahead symbol.

1.4 Construction of LL (1) Parsing Table

To construct the Parsing Table, we have two functions:

First()

If there is a variable, and from that variable if we try to drive all the strings then the beginning Terminal Symbol is called the First.

Follow()

What is the Terminal Symbol which follows a variable in the process of derivation.

Now, after computing the First and Follow set for each Non-Terminal symbol we have to con-struct the Parsing table. In the table Rows will contain the Non-Terminals and the column will contain the Terminal Symbols.

1.5 Advantages of LL (1) Parser

LL (1) parser is easy to construct and also many computer languages are designed to be LL (1).

1.6 Limitations of LL (1) Parser

An LL (1) parsing table always has conflicts when the grammar has a left-recursive pro-duction.

An LL (1) parsing table also has conflicts when the grammar contains two productions whose right-hand sides have a common non-empty prefix.

2 Design

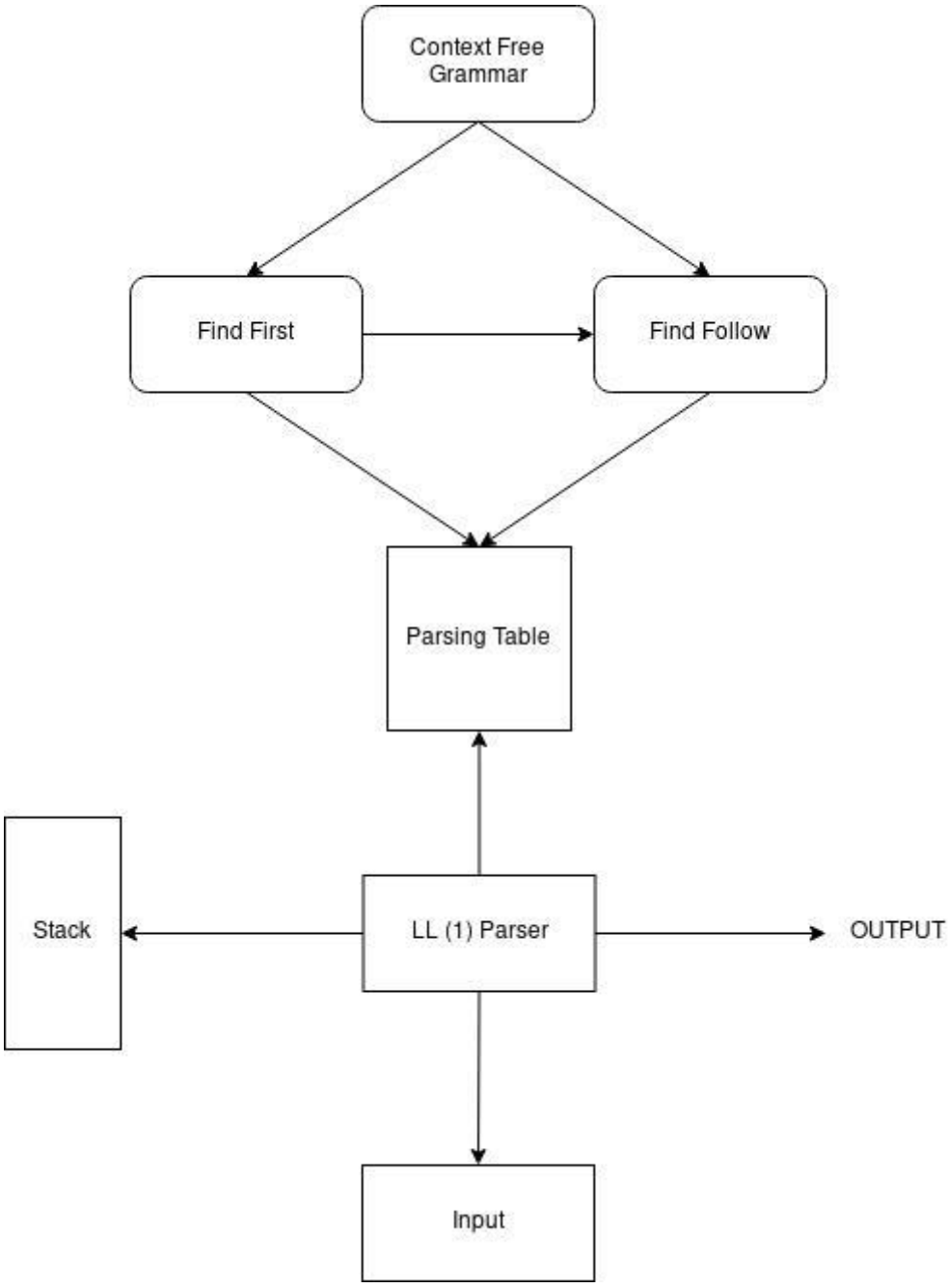

Figure 1: Design

3 Source Code

```
from tabulate import tabulate
import re
from collections import OrderedDict
import sys
import os

grammar_pattern = re.compile(r"(.+?)[=-]>n j ?(.+)")
END=’$’
EMPTY = ”
EMPTY_IN = ’’
EMPTY_OUT = ’" ’
OR = ’j ’
AND=’’

def grammar(g):
G = OrderedDict()
        def add(k, v):

                m = G.get(k, [])
        m += v
                G[k] = m

    for i, match in enumerate(grammar_pattern.finditer(g)):
            t, s2 = match.group(1), match.group(2)
              t = t.strip()
              s2 = [s.strip().split(AND) for s in s2.split(OR)]
              add(t, s2)

        K = list(G.keys())
        for i in range(len(K)):
                G[i] = G[K[i]]
        del G[K[i]]
```

```
    mp = lambda s: K.index(s) if s in K else s if s != EMPTY_IN else
    EMPTY for k in G:
    G[k] = [[mp(s) for s in n] for n in G[k]]

    return G, K


def first_set(G):

    first = dict()

    def add(k, v):
        m = first.get(k, set())
    m.add(v)
        first[k] = m

def f(k):
            r = []
            for X in G[k]:
                for x in X:
                if isinstance(x, str):
                    add(x, x)
                    r.insert(0, x)
                    if x == EMPTY:
                        add(k, x)
                    else:
                        break
                else:
                        Z = f(x)
                        for x__in f Z[i] for i in range(Z.index(EMPTY) if EMPTY
in Z else len(Z))g :
                        r.insert(0, x__)
                        add(k, x__)

    for n in r:
```

```
                add(k, n)
        return r

        for k in G:
        f(k)
        return first


def follow_set(G, S, first = None):
        first = first or first_set(G)
        follow = dict()

        def add(k, v):
                m = follow.get(k, set())
        m.add(v)
        follow[k] = m
        add(S, '$ ')

        def f(A):

        for X in G[A]:

                        for i, x in enumerate(X):
                if not isinstance(x, str):
                                fst = []
                                for y in X[i+1:]:
                                        m = first.get(y, f yg )
                                        fst += m
                                        if EMPTY not in m:
                                        break
                                        for y in fst:
                                        if y != EMPTY:
                                        add(x, y)
                                if len(fst) == 0 or EMPTY in fst:
```

```
                                    for a in follow.get(A, f g ):
                                    add(x, a)
        h = lambda f: hash(n for v in f.values() for n in v)
        f_old = h(follow)
       while True:
        for k in G:
                f(k)
        if f_old == h(follow):
                Break
        else:
                        f_old = h(follow)
        return follow

def parse_table(G, first = None, follow = None):
        first = first or first_set(G)
        follow = follow or follow_set(G, 0, first)
        M = dict()

        def add(N, k, F, t):
                m = M.get(N, dict())
                hm = m.get(k, set())
                hm.add((F, t))
                m[k] = hm
                M[N] = m
        def frst(a):

                frs = set()
                for x in a:
                        f = first[x]
                        frs = frs.union(f)
                        if EMPTY not in f:
                        break
```

```
                return frs - f EMPTYg

    def P(A):

        for i, X in enumerate(G[A]):
            for a in frst(X):
            add(A, a, A, i)
            for x in X:
            if isinstance(x, str):
                    if x == EMPTY:
                        for b in follow[A]:
                        add(A, b, A, i)
                    else:
                        break
    for g in G:
    P(g)
    return M

def as_table(G, N, M = None, first = None, follow = None):
    first = first or first_set(G)
    follow = follow or follow_set(G, 0, first)
    terminals = set(n for v in first.values() for n in v).union(set(n for v in follow.values() for
n in v))
    terminals -= f EMPTYg
    nonterminals = set(v for v in follow.keys())
    M = M or parse_table(G, first, follow)
    s = lambda s: s if s != EMPTY else
    EMPTY_OUT z = sorted(list(terminals))
    z.reverse()
    if len(z) > 4:
        z[4], z[3] = z[3], z[4]
    data = []
    for n in nonterminals:
    data2 = [N[n]]
```

```
        for t in z:
                l3 = ""
                if M.get(n) and M[n].get(t):
                for i, p in enumerate(M[n][t]):
                        A, x = p
                                x = "".join([s(e) if isinstance(e, str) else N[e] for e in G[A][x]])
                        if i > 0:
                                l3 += "n n"
                        l3 += ("% -2s ! % -2s" % (N[A], x))
                data2.append(l3)
        data2.append(" ".join(s(c) for c in first[n]))
        data2.append(" ".join(s(c) for c in follow[n]))
        data.append(data2)
        return tabulate(data, headers=["NON -n nTERMINALS"] + z+["FIRST",
"FOLLOW"], tablefmt="fancy_grid")

if __name__== "__main__":
        l = len(sys.argv)
        g = """
        E->TE'
        E' -> + T E' j
        T->FT'
        T' ->    F T' j
        F -> ( E ) j id
        """
        if l >= 2:
                arg = sys.argv[1]
        if arg in ("–help", "-h"):
                        print("Execution Command : ")
                print("python3 parsing_table.py <input_file>")
        elif os.path.isfile(arg):
                        with open(arg, "r") as file:
                G, N = grammar(file.read())
                        if len(G) == 0:
```

```
                    print("Invalid grammar in file.")
                    exit(-1)
elif re.match(grammar_pattern, arg):
            G, N = grammar(arg)
    else:
              print("No grammar was found.")
        exit(-1)
    Else:
    print("Example parse table:")
    G, N = grammar(g)
    try:
        table = as_table(G, N)
    print(table)
    except RecursionError:
        print("Left recursive grammar!")
```

4 Execution Details

Insert valid grammar in grammar.txt file.

Run parsing_table.py with grammar.txt file as

parameter $ python3 parsing_table.py grammar.txt

5 Output

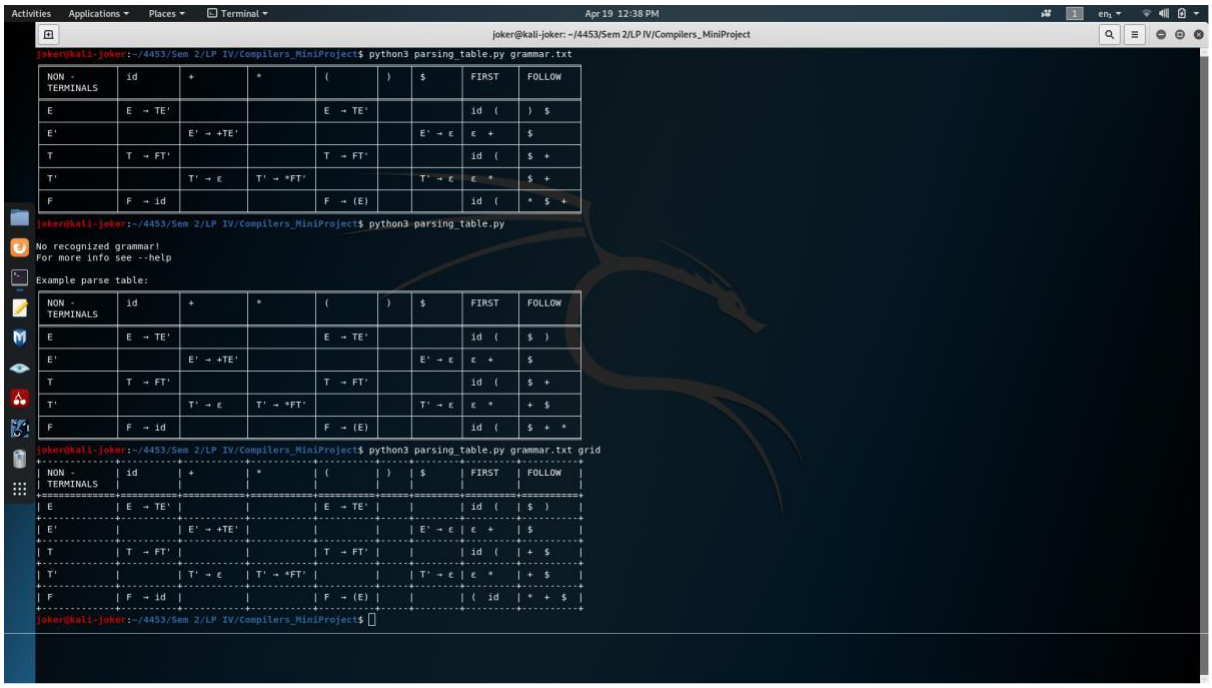

Figure 2: Output

6 Conclusion and future scope

6.1 Conclusion

To conclude, we implemented the LL (1) parsing table generation algorithm which can be further used by LL (1) parser to parse the input stream.

6.2 Future Scope

The future enhancement would be,

To remove left recursion within the grammar, if any.

Implementing the LL (1) parser to parse the given input stream.