LL(1) Parser Simulator

MINI PROJECT REPORT

18CSC304J – Compiler Design

LABORATORY

(2018 Regulation)

III Year/ VI Semester

Academic Year: 2022 -2023

By

S.M.DINESH(RA2011028010057)

KARTHI C(RA2011028010072)

DOMA BHARGAV(RA2011028010069)

Under the guidance of

Dr. C. Fancy

Assistant Professor

Department of Networking and Communications

DEPARTMENT OF NETWORKING AND COMMUNICATIONS

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Kancheepuram

MAY 2023

BONAFIDE

Certified that this Mini project report titled **LL(1) Parser Simulator** for the course **18CSC304J– Compiler Design** is the bonafide work of**S.M.DINESH (RA2011028010057), KARTHI C(RA2011028010072) and DOMA BHARGAV(RA2011028010069)** who undertook the task of completing the project within the allotted time.

Signature

Dr. C. Fancy

Course Faculty
Assistant Professor
Department of NWC

Signature

Dr. Annapurani Panaiyappan K

Head of the Department
Professor
Department of NWC

TOPIC: - LL(1) Parser Simulator

Department	**Computer Science in Cloud Computing (NWC)**
Subject	**Compiler Design**
Semester	**6**
Faculty	**Dr. C. Fancy**
Submitted by	**S.M.DINESH(RA2011028010057)** **KARTHI C(RA2011028010072)** **DOMA BHARGAV(RA2011028010069)**

ABSTRACT

An LL(1) parser is a type of top-down parser that uses a lookahead of one token to parse a given input. It is a type of predictive parser that can parse any LL(1) grammar, where "LL" stands for left-to-right scan of the input, and "1" means a lookahead of one token. The LL(1) parser simulator is a program that takes as input an LL(1) grammar and an input string, and simulates the parsing process using the steps above to construct a parse tree for the input string.

The LL(1) parser is a parsing algorithm used to parse context-free grammars. It is a top-down parsing method that starts with the grammar's start symbol and generates a parse tree by applying production rules to non-terminals and matching tokens in the input. The LL(1) parser is known for its simplicity, efficiency, and deterministic behavior.The LL(1) parser is based on a predictive parsing table that maps a pair of a non-terminal and a lookahead symbol to a production rule.

The table is constructed by analyzing the grammar for left-factoring and left-recursion elimination, computing the first and follow sets of each non-terminal, and constructing a parsing table based on these sets.The LL(1) parser is widely used in practice and is often implemented as part of a compiler or interpreter. It is capable of parsing a wide range of context-free grammars and is particularly useful for parsing programming languages.

Table of Contents

INTRODUCTION

The LL Parser(1) project is a software development project that aims to create a parser using the LL(1) parsing algorithm. LL(1) is a top-down parsing algorithm that parses a string of tokens from left to right, using a leftmost derivation without backtracking.The LL(1) parsing algorithm is a deterministic algorithm that is commonly used for parsing programming languages. It is widely used because it is efficient, simple to implement, and can handle a wide range of grammars.The LL Parser(1) project involves several stages, including designing the grammar of the language to be parsed, implementing a lexer to tokenize the input, constructing a parse table, and finally, implementing the LL(1) parsing algorithm itself.The resulting parser will be capable of analyzing the syntax of input programs and identifying any syntax errors. It will also be able to generate a parse tree, which can be used for further analysis, such as semantic analysis and code generation.

Problem Statement:

To design and implement a program that takes as input a LL(1) grammar and a string to parse, and produces as output either an error message indicating that the input string is not in the language of the grammar or a parse tree that represents the structure of the input string according to the grammar.The LL(1) parser simulator must also handle situations where the input string is not in the language of the grammar, such as when there is no matching entry in the parsing table or when there is a syntax error in the input string. In these cases, the simulator should report an appropriate error message to the user.

Objective:

- By simulating an LL(1) parser, one can verify whether a given grammar is LL(1) or not, and whether a given input string can be parsed by the LL(1) parsing algorithm. This is useful for compiler designers, who need to ensure that their grammar is unambiguous and can be parsed efficiently by atop-down parser.

- LL(1) parser simulator project would typically involve creating a software application that can read a context-free grammar (CFG) and simulate the parsing process of the input language based on the LL(1) parsing algorithm

- The main objective of the LL(1) parser is to parse context-free grammars and generate a parse tree for a given input string. More specifically, the objectives of an LL(1) parser include:

- In order to achieve these objectives, an LL(1) parser must be able to analyze the grammar, construct a parsing table based on the grammar, and use the parsing table to parse the input string. The parsing process involves repeatedly matching the input tokens with the production rules in the parsing table until a complete parse tree is generated or a syntax error is encountered.

- Overall, the objective of an LL(1) parser is to provide a reliable and efficient method for parsing context-free grammars, which is an essential component in the development of software tools for processing programming languages.

Literature Study:

There have been several studies on LL(1) parser algorithm and its implementation. Many researchers have proposed various techniques to improve the efficiency and accuracy of LL(1) parsing. For example, some researchers have proposed a technique called LL(*) parsing, which can handle more complex grammars than LL(1) parsing.Others have proposed techniques to handle left-recursion and left-factorization in LL(1) parsing.

The LL(1) parsing algorithm is a widely used technique in computer science and is covered in many textbooks and academic papers. Here are some notable sources for a literature study of LL(1) parsers:

1. "Compilers: Principles, Techniques, and Tools" by Aho, Sethi, and Ullman - This classic textbook provides a thorough introduction to compiler design and includes an in-depth discussion of LL(1) parsing algorithms.

2. "Parsing Techniques: A Practical Guide" by Grune and Jacobs - This book provides a comprehensive overview of parsing techniques and includes a chapter dedicated to LL(1) parsing.

3. "An Efficient Method of LL(1) Parsing" by DeRemer and Pennello - This seminal paper introduced the concept of LL(1) parsing and provides a detailed algorithm for constructing an LL(1) parser.

4. "Parsing Theory: Volume 1, Languages and Parsing" by Sippu and Soisalon-Soininen - This book covers the theory behind parsing algorithms and includes a chapter dedicated to LL(1) parsing.

The LL(1) parsing algorithm is a widely used technique in computer science and is covered in many textbooks and academic papers. Here are some notable sources for a literature study of LL(1) parsers:

5. "Introduction to Compiler Construction in a Java World" by Mössenböck and Würthinger - This book provides a practical introduction to compiler

design and includes a detailed discussion of LL(1) parsing algorithms using Java.

6. "The Dragon Book" by Aho, Lam, Sethi, and Ullman - This classic textbook covers compiler design and includes a chapter on LL(1) parsing algorithms.

7. "Programming Language Processors in Java: Compilers and Interpreters" by Watt and Brown - This book provides a practical introduction to compiler design and includes a detailed discussion of LL(1) parsing algorithms using Java.

These sources provide a comprehensive understanding of LL(1) parsing algorithms, including both theory and practical implementation details.

FRONTEND CODE:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>LL(1) Parser Simulator</title>
    <link rel="stylesheet" href="./style.css">
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto&display=swap"
rel="stylesheet">
</head>

<body>
    <div class="sets">
        <h4>Generate First/Follow/Predict Sets</h4>
        <button onclick="addRule()">Add More Rule</button>
        <button onclick="generateSets()">Generate Sets</button>
        <div id="rules_tbl"></div>
        <div id="fsls_tbl"></div>
        <div id="ps_tbl"></div>
    </div>

    <div class="tests">
        <h4>Parse Input String</h4>
        <button onclick="parse()">Parse Input</button>
        <div id="tests_tbl"></div>
        <div id="tree"></div>
    </div>

    <script type="text/javascript" src="https://unpkg.com/tabulator-
tables@4.6.2/dist/js/tabulator.min.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/raphael/2.3.0/raphael.min.js"></script
>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/treant-
js/1.0/Treant.min.js"></script>
    <!-- partial -->
    <script src="./script.js"></script>

</body>

</html>
```

```css
@import url("https://unpkg.com/tabulator-
tables@4.6.2/dist/css/tabulator.min.css");
@import url("https://cdnjs.cloudflare.com/ajax/libs/treant-js/1.0/Treant.css");

body {
  font-family: "Roboto", sans-serif;
  text-align: center;
}
button {
  margin-bottom: 8px;
}

.sets,
.tests {
  display: inline-block;
  margin: 0 8px;
  width: 45%;
  vertical-align: top;
}

@media (max-width: 800px) {
  .sets,
  .tests {
    width: 90%;
  }
}
```

BACKEND CODE :

```
// LL(1) parser code

let getStartSymbol = ([start, rules]) => start;

let getNonterminals = ([start, rules]) =>
    rules.reduce((a, [lhs, rhs]) => [...a, lhs], []);

let getInitFirstSets = (g) =>
    getNonterminals(g).reduce((m, nt) => m.set(nt, new Set()), new Map());

let getInitFollowSets = (g) =>
    getInitFirstSets(g).set(getStartSymbol(g), new Set(['eof']));

let computeFirstSet = (firstSet, [h, ...t]) => {
    let first = mapClone(firstSet);
    if (h == undefined) return new Set(['eps']);
    if (first.get(h)) {
        let h_first = first.get(h);
        if (h_first.has('eps')) {
            h_first.delete('eps');
            return new Set([...computeFirstSet(first, t), ...h_first]);
        } else return h_first;
    }
    return new Set([h]);
};

let recurseFirstSets = ([start, rules], firstSet, firstFunc) => {
    let first = mapClone(firstSet);
    return rules.reduce(
        (map, [lhs, rhs]) => map.set(
            lhs, new Set([...firstFunc(first, rhs), ...first.get(lhs)])),
        first
    );
};

let mapClone = (m) => {
    let clone = new Map();
    for (key of m.keys()) {
        clone.set(key, new Set(m.get(key)));
    }
    return clone;
};

let getFirstSets = (g, first, firstFunc) => {
    let setEquals = (s1, s2) => (s1.size == s2.size) &&
[...s1].every(v=>s2.has(v));
    let mapEquals = (m1, m2) => [...m1.keys()].every(key => setEquals(m1.get(key),
m2.get(key)));
    let updated = recurseFirstSets(g, first, firstFunc);
```

```
    if (mapEquals(first, updated)) {
        return updated;
    } else {
        return getFirstSets(g, updated, firstFunc);
    }
};

let updateFollowSet = (firstSet, followSet, nt, [h, ...t]) => {
    // JS pass by reference, MUST deep clone params for immutability principal
    let first = mapClone(firstSet);
    let follow = mapClone(followSet);
    if (h == undefined) return follow;
    if (first.get(h)) {
        if (t.length > 0) {
            let tail_first = computeFirstSet(first, t);
            if (tail_first.has('eps')) {
                tail_first.delete('eps');
                let updated = follow.set(h, new Set(
                    [...follow.get(h), ...tail_first, ...follow.get(nt)]
                ));
                return updateFollowSet(first, updated, nt, t);
            } else {
                let updated = follow.set(h, new Set(
                    [...follow.get(h), ...tail_first]
                ));
                return updateFollowSet(first, updated, nt, t);
            }
        }
        return follow.set(h, new Set(
            [...follow.get(h), ...follow.get(nt)]
        ));
    }
    return updateFollowSet(first, follow, nt, t);
};

let recurseFollowSets = ([start, rules], first, follow, followFunc) =>
rules.reduce((map, [lhs, rhs]) => followFunc(first, map, lhs, rhs), follow);

let getFollowSets = (g, first, follow, followFunc) => {
    let setEquals = (s1, s2) =>
        (s1.siz == s2.size) && [...s1].every(v=>s2.has(v));
    let mapEquals = (m1, m2) =>
        [...m1.keys()].every(key => setEquals(m1.get(key), m2.get(key)));
    let updated = recurseFollowSets(g, first, follow, followFunc);
    if (mapEquals(follow, updated))
        return getFollowSets(g, first, updated, followFunc);
    else
        return updated;
};
```

```
let getPredictSets = ([start, rules], first, follow, firstFunc) =>
rules.reduce((l, [lhs, rhs]) => {
    let rhs_first = firstFunc(first, rhs);
    if (rhs_first.has('eps')) {
        rhs_first.delete('eps');
        return [
            [[lhs, rhs], new Set([...rhs_first, ...follow.get(lhs)])],
            ...l
        ];
    } else {
        return [
            [[lhs, rhs], firstFunc(first, rhs)],
            ...l
        ];
    }
}, []);

let getPredictMap = predictSets => predictSets.reduce(
    (map, [[lhs, rhs], symbols]) => [...symbols].reduce((pmap, symbol)
=>pmap.set(lhs+'|||'+symbol, rhs), map),
    new Map()
);

let derive = (first, predict, [ph, ...pt], [ih, ...it]) => {
    if (ph == undefined) {
        if (ih == undefined) return true;
        return false;
    } else {
        if (first.get(ph)) {
            if (ih == undefined) return false;
            let expanded = predict.get(ph+'|||'+ih);
            if (expanded == undefined) return false;
            return derive(first, predict, [...expanded, ...pt], [ih, ...it]);
        } else {
            if (ph == ih) return derive(first, predict, pt, it);
            return false;
        }
    }
};

let tryDerive = (g, inputStr) => {
    let ifs = getInitFirstSets(g);
    let fs = getFirstSets(g, ifs, computeFirstSet);
    let ifl = getInitFollowSets(g);
    let ls = getFollowSets(g, fs, ifl, updateFollowSet);
    let pset = getPredictSets(g, fs, ls, computeFirstSet);
    let pmap = getPredictMap(pset);
    return derive(fs, pmap, [getStartSymbol(g), "eof"], [...inputStr, 'eof']);
};
```

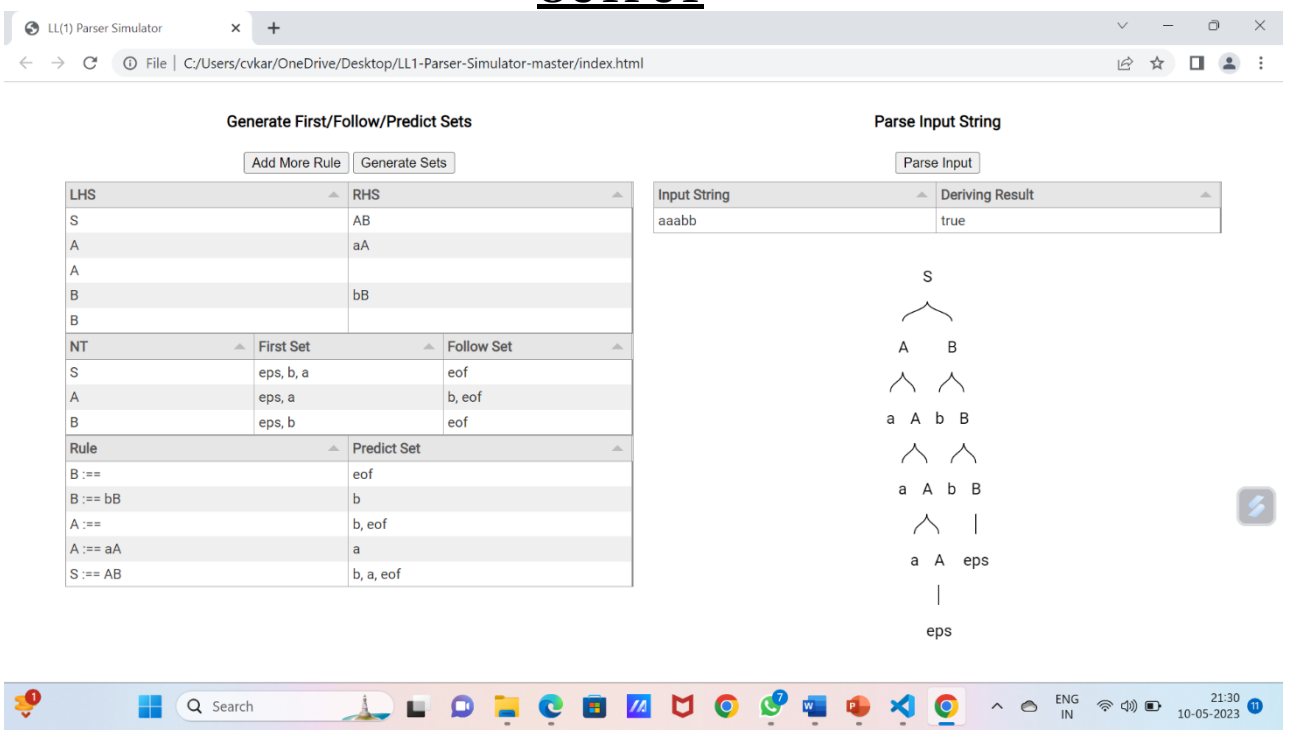
```
let deriveTree = (first, predict, symbol, input) => {
    if (first.get(symbol)) {
        let [ih, ...it] = input;
        if (ih == undefined) {
            throw new Error("Invalid input");
        } else {
            let expanded = predict.get(symbol+'|||'+ih);
            if (expanded == undefined) {
                throw new Error("Invalid input");
            } else {
                if (expanded.length == 0) {
                    return [
                        { text: { name: symbol }, children: [ { text: {name:
'eps'} } ] },
                        input
                    ];
                } else {
                    let [eh, ...et] = expanded;
                    let createTree = ([subTree, input], symbol) => {
                        let [childTree, newInput] = deriveTree(first, predict,
symbol, input);
                        return [[...subTree, childTree], newInput];
                    };
                    let [expandedTree, newInput] = expanded.reduce(
                        createTree, [[], input]);
                    return [
                        { text: {name: symbol}, children:  expandedTree },
                        newInput
                    ];
                }
            }
        }
    } else {
        let [ih, ...it] = input;
        if (ih == undefined) {
            throw new Error("Invalid input");
        } else {
            return [{ text: { name: ih } }, it];
        }
    }
};

let tryDeriveTree = (g, inputStr) => {
    let ifs = getInitFirstSets(g);
    let fs = getFirstSets(g, ifs, computeFirstSet);
    let ifl = getInitFollowSets(g);
    let ls = getFollowSets(g, fs, ifl, updateFollowSet);
    let pset = getPredictSets(g, fs, ls, computeFirstSet);
    let pmap = getPredictMap(pset);
```

```
    let [tree, input] = deriveTree(fs, pmap, getStartSymbol(g), [...inputStr,
'eof']);
    return tree;
};

// UI Code

var rules = [
    {lhs: 'S', rhs: 'AB' },
    {lhs: 'A', rhs: 'aA' },
    {lhs: 'A', rhs: '' },
    {lhs: 'B', rhs: 'bB' },
    {lhs: 'B', rhs: '' }
];

var cases = [
    { case: 'aaabb', result: '' }
];

var rules_tbl = new Tabulator('#rules_tbl', {
    layout: 'fitColumns',
    data: rules,
    reactiveData: true,
    addRowPos: 'bottom',
    columns:[
        {title: 'LHS', field: 'lhs', editor: true},
        {title: 'RHS', field: 'rhs', editor: true}
    ],
});

var tests_tbl = new Tabulator('#tests_tbl', {
    layout: 'fitColumns',
    data: cases,
    reactiveData: true,
    addRowPos: 'bottom',
    columns:[
        { title: 'Input String', field: 'case', editor:true },
        { title: 'Deriving Result', field: 'result' }
    ],
});

var sets_tbl = new Tabulator('#fsls_tbl', {
    layout: 'fitColumns',
    reactiveData: true,
    columns:[
        { title: 'NT', field: 'nt' },
        { title: 'First Set', field: 'first' },
        { title: 'Follow Set', field: 'follow' }
    ],
});
```

```
var ps_tbl = new Tabulator('#ps_tbl', {
    layout: 'fitColumns',
    reactiveData: true,
    columns:[
        { title: 'Rule', field: 'rule' },
        { title: 'Predict Set', field: 'predicts' }
    ],
});

let getGrammar = () => {
    let g = [];
    let rows = rules_tbl.getRows();
    rows.forEach(r=> {
        if (r.getData().lhs != undefined) {
            if (r.getData().rhs == undefined)
                g.push([r.getData().lhs, []]);
            else
                g.push([r.getData().lhs, [...r.getData().rhs]]);
        }
    });
    return ['S', g];
};

let getTests = () => {
    let tests= [];
    for (r of tests_tbl.getRows()) {
        if (r.getData().case == undefined)
            tests.push({case: '', result: ''});
        else
            tests.push(r.getData());
    }
    return tests;
}

let generateSets = () => {
    let g = getGrammar();
    let ifs = getInitFirstSets(g);
    let fs = getFirstSets(g, ifs, computeFirstSet);
    let ifl = getInitFollowSets(g);
    let ls = getFollowSets(g, fs, ifl, updateFollowSet);
    let pset = getPredictSets(g, fs, ls, computeFirstSet);

    let fsls_data = [];
    for (nt of fs.keys()) {
        fsls_data.push({nt: nt, first: [...fs.get(nt)].join(', '), follow:
[...ls.get(nt)].join(', ')});
    }
    sets_tbl.setData(fsls_data);
```

```
    let ps_data = [];
    for (r of pset) {
        ps_data.push({rule: r[0][0]+" :== "+[...r[0][1]].join(''), predicts:
[...r[1]].join(', ')});
    }
    ps_tbl.setData(ps_data);
};

function parse() {
    let g = getGrammar();
    let cases= getTests();
    for (let i=0; i<cases.length; i++) {
        cases[i].result = tryDerive(g, [...cases[i].case]);
    }
    tests_tbl.setData(cases);

    // generate tree graph ONLY when first test case is valid input
    let tree_config = {
        chart: {
            container: "#tree",
            levelSeparation: 20,
            siblingSeparation: 15,
            subTeeSeparation:  15,
            rootOrientation: "NORTH"
        },
        nodeStructure: {}
    };
    if (cases[0].result) {
        tree_config.nodeStructure =
            tryDeriveTree(g, [...cases[0].case]);
    } else {
        tree_config.nodeStructure = {};
    }
    // draw tree
    new Treant( tree_config );
};

let addRule = () => { rules_tbl.addRow(); };
let addTest = () => { tests_tbl.addRow(); };

generateSets();
parse();
setTimeout(function(){ parse(); },5000);
```

OUTPUT


LL(1) Parser Simulator
File | C:/Users/cvkar/OneDrive/Desktop/LL1-Parser-Simulator-master/index.html
Generate First/Follow/Predict Sets
Add More Rule
Generate Sets
LHS
RHS
S
AB
A
aA
A
B
bB
B
NT
First Set
Follow Set
S
eps, b, a
eof
A
eps, a
b, eof
B
eps, b
eof
Rule
Predict Set
B :==
eof
B :== bB
b
A :==
b, eof
A :== aA
a
S :== AB
b, a, eof
Parse Input String
Parse Input
Input String
Deriving Result
aaabb
true
S
A B
a A b B
a A b B
a A eps
eps
Search
ENG
IN
21:30
10-05-2023

Conclusion:

LL(1) parser has some limitations, such as the inability to handle left-recursive grammars and ambiguous grammars. It also requires the grammar to be LL(1), meaning it must be unambiguous, and there must be at most one production rule for each non-terminal and lookahead symbol combination.Despite these limitations, LL(1) parser is widely used in practice because it is easy to implement, efficient, and can handle a large class of grammars. It is commonly used in compilers and other software tools that need to analyze and parse programming languages. In conclusion, LL(1) parser is a powerful parsing technique that has its limitations but is still widely used in practice. It provides an efficient and straightforward way to parse input strings based on a given grammar, making it a valuable tool for software development.

References:

- https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/amp/

- https://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm

https://unacademy.com/lesson/ll1-parser-with-example/ZKRUVPVE