College of Engineering Trivandrum

Compiler Design Lab

Abhishek Manoharan
S7 CSE Roll No:2

TVE17CS002

Department of Computer Science

January 18, 2021

CS431 - Compiler Design Lab · 2020 ·

Final Exam Report

1 LL(1) Parser

1.1 Question

For the Grammar G given, build an LL (1) parser and check for the acceptance of any given string. If the given grammar is Left Recursive, write the code for removing it. Input:
(1) Set of terminals, set of non-terminals, productions and start symbol of the left recursive grammar
(2) Input string to be parsed

1.2 Theory

LL(1)

LL (1) is a Top-down parser, where the 1st L represents that the scanning of the Input will be done from Left to Right manner and second L shows that in this Parsing technique we are going to use Left most Derivation Tree. and finally the 1 represents the number of look ahead, means how many symbols are you going to see when you want to make a decision. Non- Recursive predictive parsing is a table-driven parser.

1.3 Algorithm

Algorithm 1: Algorithm for Shift Reduce parser

```
for each production rule A -> a of a grammar G
     for each terminal a in FIRST(a)    add A    a to M[A,a]
     If e in FIRST(a)    for each terminal a in FOLLOW(A) add A    a to M[A,a]
     If e in FIRST(a) and $ in FOLLOW(A)    add A    a to M[A,$]
     All other undefined entries of the parsing table are error entries.

If X and a are $    parser halts (successful completion)
    If X and a are the same terminal symbol (different from $)    parser pops X from the
stack, and moves the next symbol in the input buffer.
    If X is a non-terminal    parser looks at the parsing table entry M[X,a]. If M[X,a] holds
a production rule X ->Y1Y2...Yk, it pops X from the stack and pushes Yk,Yk-1,...,Y1 into
the stack. The parser also outputs the production rule X->Y1Y2...Yk to represent a step
of the derivation.
    None of the above    error
    All empty entries in the parsing table are errors.
    If X is a terminal symbol different from a, this is also an error case.
```

1.4 Code

```
#include <bits/stdc++.h>
using namespace std;
vector<vector<string>> get_production(unordered_map<char, int> &non_term, int *num)
{
    int non = -1;
    string s;
    vector<vector<string>> production(100);
    getline(cin, s);
    while (s != "")
    {
        int non_index;
        char left = s[0];
        if (non_term.find(s[0]) == non_term.end())
        {
            non_term[s[0]] = ++non;
            non_index = non;
        }
        else
        {
```

```
            non_index = non_term[s[0]];
        }
        string right = s.substr(4, s.size() - 4);
        production[non].push_back(right);
        //cout << "right side " << right << endl;
        getline(cin, s);
    }
    *num = non;
    return production;
}
unordered_set<char> split_string(string s)
{
    int n = s.size();
    unordered_set<char> result;
    for (int i = 0; i < n; ++i)
    {
        if (s[i] != ' ')
        {
            result.insert(s[i]);
        }
    }
    return result;
}
void print_stack(stack<char> check)
{
    string s = "";
    while (!check.empty())
    {
        s = check.top() + s;
        check.pop();
    }
    cout << s;
}
vector<vector<string>> remove_left(vector<vector<string>> production, unordered_map<char, int>
     &non_term, int *nn)
{
    char extra = 'Z';
    int non = *nn;
    //cout << non << " is non terminal count" << endl;
    vector<vector<string>> no_left(10);
    for (int i = 0; i < production.size(); ++i)
    {
        int flag = 0;
        for (int j = 0; j < production[i].size(); ++j)
        {
            if (non_term.find(production[i][j][0]) != non_term.end() && non_term[production[i
    ][j][0]] == i)
            {
                flag = 1;
                cout << "left recursion found in " << production[i][j] << endl;
                vector<string> new_prod;
                string alpha = production[i][j].substr(1, production[i][j].size() - 1);
                cout << "alpha is " << alpha << endl;
                for (int k = 0; k < production[i].size(); ++k)
                {
                    string rhs = "";
                    if (k != j)
                    {
                        rhs = production[i][k] + extra;
                        new_prod.push_back(rhs);
                        cout << "first production push back" << endl;
                    }
                }
                no_left[i] = new_prod;
                vector<string> second_prod;
                second_prod.push_back(alpha + extra);
                second_prod.push_back("#");
                cout << "Extra prodcution pushing back" << endl;
                no_left[++non] = second_prod;
                non_term[extra] = non;
                extra = extra - 1;
                cout << "push back ended" << endl;
            }
        }
        if (flag == 0)
        {
```

```
            for (int m = 0; m < production[i].size(); ++m)
                no_left[i].push_back(production[i][m]);
        }
    }
    *nn = non;
    // cout << "new number of non terminal is : " << non << endl;
    // for (auto x : no_left)
    // {
    //     for (auto y : x)
    //     {
    //         cout << y << "\t";
    //     }
    //     cout << endl;
    // }
    return no_left;
}
vector<char> find_first(char c, vector<vector<string>> production, vector<vector<char>> &First
    , unordered_map<char, int> umap)
{
    vector<char> res;
    if (umap.find(c) == umap.end())
    {
        res.push_back(c);
        return res;
    }
    int num = umap[c];
    if (First[num].size() != 0)
    {
        return First[num];
    }
    int n = production[num].size();
    for (int i = 0; i < n; ++i) // iterate through each production
    {
        string m = production[num][i];
        int right_size = m.size();
        for (int j = 0; j < right_size; ++j) // iterate through each charecter in production
        {
            if (umap.find(m[j]) == umap.end()) //if right side of production  is a terminal
            {
                if (find(res.begin(), res.end(), m[j]) == res.end())
                {
                    res.push_back(m[j]);
                }

                break;
            }
            else // Non terminal
            {
                vector<char> temp = find_first(m[j], production, First, umap); // finding
    first of  j th non terminal
                //cout << "called for first of " << m[j] << endl;
                int first_char = temp.size();
                int flag = 1;
                for (int k = 0; k < first_char; ++k)
                {
                    if (temp[k] == '#')
                    {
                        // cout << "Epsilon found in first of " << m[j] << endl;
                        flag = 0;
                    }
                    if (find(res.begin(), res.end(), temp[k]) == res.end())
                    {
                        if (temp[k] != '#')
                        {
                            res.push_back(temp[k]);
                        }
                        else
                        {
                            if (j == right_size - 1)
                            {
                                res.push_back(temp[k]);
                            }
                        }
                    }
                }
                if (flag == 1)
```

```
                {
                    break;
                }
            }
        }
    }
    First[num] = res;
    return res;
}
unordered_set<char> find_follow(char c, vector<vector<string>> production, vector<
    unordered_set<char>> &follow, unordered_map<char, int> umap, vector<vector<char>> first)
{
    if (!follow[umap[c]].empty())
    {
        return follow[umap[c]];
    }
    //cout << "called follow of " << c << endl;
    unordered_set<char> res;
    if (umap[c] == 0)
    {
        //cout << "added $ in follow of " << c << endl;
        res.insert('$');
    }
    int n = production.size();
    for (int i = 0; i < n; ++i)
    {
        for (auto x : production[i])
        {                       // considering each production
            int m = x.size(); //read rhs charecter by charecter
            for (int j = 0; j < m; ++j)
            {
                if (x[j] == c) // if we find charecter in right side of production
                {
                    //cout << c << " found in production " << x << endl;
                    if (j == m - 1)
                    { //last element
                        //  cout << c << " is the edning charecter" << endl;
                        char check;
                        for (auto y : umap)
                        {
                            if (y.second == i)
                            {
                                check = y.first;
                            }
                        }
                        if (check != c)
                        {
                            unordered_set<char> sample = find_follow(check, production, follow
    , umap, first);
                            for (auto y : sample)
                            {
                                //cout << y << " inserted in follow of " << c << endl;

                                res.insert(y);
                            }
                            //cout << endl;
                        }
                    }
                    else
                    {
                        for (int k = j + 1; k < m; ++k)
                        {
                            int flag = 1;
                            if (umap.find(x[k]) == umap.end())
                            { // checking whether char is termi if so add and stop
                                // cout << "since found non terminal " << x[k] << "stop here
    added it "
                                //<< "in follow of " << c << endl;
                                res.insert(x[k]);
                                flag = 1; //should stop here
                            }
                            else
                            { // if it is a non terminal then add its first
                                int first_b = first[umap[x[k]]].size();
                                for (int l = 0; l < first_b; ++l)
                                {
```

```
                                    if (first[umap[x[k]]][l] != '#')
                                    {
                                        res.insert(first[umap[x[k]]][l]);
                                        //cout << first[umap[x[k]]][l] << " Added to follow of
     " << c << endl;
                                    }
                                    else //we found an epsilon in first so process should
    continues
                                    {
                                        flag = 0;
                                        if (k == m - 1) // first[b] has #
                                        {
                                            char check;
                                            for (auto y : umap)
                                            {
                                                if (y.second == i)
                                                {
                                                    check = y.first;
                                                }
                                            }
                                            if (check != c)
                                            {
                                                unordered_set<char> sample = find_follow(check
    , production, follow, umap, first);
                                                for (auto y : sample)
                                                {
                                                    //cout << y << "added to follow of " << c
    << endl;
                                                    res.insert(y);
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                            if (flag == 1)
                            {
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
    return res;
}
void print_star(int n)
{
    for (int i = 0; i < n; ++i)
    {
        cout << "*";
    }
    cout << endl;
}
int main()
{
    int non = -1;
    string s;
    vector<vector<string>> production(100);
    cout << "Enter the productions in the form \"S : r\" " << endl;
    unordered_map<char, int> non_term;
    production = get_production(non_term, &non);
    unordered_set<char> terminals;
    unordered_set<char> non_terminals;
    cout << "Non-terminals: ";
    getline(cin, s);
    non_terminals = split_string(s);
    // for (auto x : non_terminals)
    // {
    //     cout << x << " ";
    // }
    // cout << endl;
    cout << "Terminals: ";
    getline(cin, s);
    terminals = split_string(s);
```

```
    vector<vector<string>> new_production(100);
    new_production = remove_left(production, non_term, &non);
    vector<vector<char>> First(non + 1);
    cout << "New grammer after removing left recursion" << endl;
    print_star(20);
    //cout << "non terminal count is : " << non + 1 << endl;
    for (auto x : non_term)
    {
        if (non_terminals.find(x.first) == non_terminals.end())
        {
            non_terminals.insert(x.first);
        }
    }
    for (int i = 0; i <= non; ++i)
    {
        char left;
        for (auto x : non_term)
        {
            if (x.second == i)
            {
                left = x.first;
                //cout << x.first << " --> ";
            }
        }
        for (int j = 0; j < new_production[i].size(); ++j)
        {
            cout << left << " --> " << new_production[i][j] << "\t";
        }
        cout << endl;
    }
    for (auto x : non_terminals)
    {
        First[non_term[x]] = find_first(x, new_production, First, non_term);
    }
    cout << "-----First----- " << endl;
    for (auto x : non_terminals)
    {
        cout << x << ": ";
        for (auto y : First[non_term[x]])
        {
            cout << y << " ";
        }
        cout << endl;
    }
    vector<unordered_set<char>> follow(non + 1);
    cout << "Enter the Start symbol: ";
    char c;
    cin >> c;
    cout << "-----Follow-----" << endl;
    for (auto x : non_terminals)
    {
        if (follow[non_term[x]].empty())
            follow[non_term[x]] = find_follow(x, new_production, follow, non_term, First);
    }
    for (auto x : non_terminals)
    {
        cout << x << ": ";
        for (auto y : follow[non_term[x]])
        {
            cout << y << " ";
        }
        cout << endl;
    }
    int terminal_count = terminals.size();
    //cout << "terminal count is : " << terminal_count << endl;
    vector<vector<string>> parse_table(100, vector<string>(100));
    unordered_map<char, int> term_mapp;
    int ter_count = 0;
    for (auto x : terminals)
    {
        term_mapp[x] = ter_count++;
        //cout << x << endl;
    }
    for (int i = 0; i <= non; ++i)
    {
        for (int j = 0; j < new_production[i].size(); ++j)
```

```
        {
            for (int k = 0; k < new_production[i][j].size(); ++k)
            {
                int flag = 0;
                if (terminals.find(production[i][j][k]) != terminals.end())
                {
                    if (production[i][j][k] != '#')
                    {
                        parse_table[i][term_mapp[production[i][j][k]]] = production[i][j];
                        //cout << production[i][j] << " added to " << i << production[i][j][k]
    << endl;
                        break;
                    }
                    else
                    {
                        flag = 1;
                    }
                }
                else
                {
                    vector<char> its_first = First[non_term[production[i][j][k]]];
                    for (int m = 0; its_first.size(); ++m)
                    {
                        if (its_first[m] != '#')
                            parse_table[i][term_mapp[its_first[m]]] = production[i][j];
                        else
                        {
                            flag = 1;
                        }
                    }
                }
                if (flag == 1)
                {
                    unordered_set<char> its_follow = follow[i];
                    for (auto x : its_follow)
                    {
                        if (x != '$')
                        {
                            parse_table[i][term_mapp[x]] = production[i][j];
                        }
                        else
                        {
                            parse_table[i][term_mapp['#']] = production[i][j];
                        }
                    }
                }
            }
        }
    }
    cout << "Parsing table" << endl;
    cout << "\t";
    for (auto x : terminals)
    {
        if (x != '#')
            cout << x << "\t";
        else
        {
            cout << '$' << "\t";
        }
    }
    cout << endl;
    for (int i = 0; i <= non; ++i)
    {
        char left;
        for (auto x : non_term)
        {
            if (x.second == i)
            {
                left = x.first;
                //cout << x.first << " --> ";
            }
        }
        cout << left << "\t";
        for (int j = 0; j < ter_count; ++j)
        {
            cout << parse_table[i][j] << "\t";
```

```
        }
        cout << endl;
    }
    cout << "Enter the string to be parsed" << endl;
    string input;
    cin >> input;
    stack<char> parsing;
    input = input + '$';
    parsing.push('$');
    parsing.push(c);
    int keep = 0;
    cout << "stack\tInput\toutput" << endl;
    print_stack(parsing);
    cout << "\t" << input << endl;
    for (int i = 0; i < 100; ++i)
    {
        char molil = parsing.top();
        //cout << molil << " is in top " << endl;
        if (terminals.find(parsing.top()) != terminals.end())
        {
            //cout << "terminal  found" << endl;
            if (parsing.top() == input[keep])
            {
                parsing.pop();
                keep++;
                print_stack(parsing);
                cout << "\t" << input.substr(keep, input.size() - keep) << endl;
            }
            else
            {
                cout << "invalid input parsing cant be done" << endl;
                break;
            }
        }
        else
        {
            if (input[keep] == '$' && molil == '$')
            {
                cout << "parsing finished successfully" << endl;
                break;
            }
            //cout << "Non terminal in top of stack" << endl;
            string action = parse_table[non_term[parsing.top()]][term_mapp[input[keep]]];
            if (action == "")
            {
                cout << "invalid input no productuon present" << endl;
                break;
            }
            else
            {
                if (action == "#")
                {
                    parsing.pop();
                    print_stack(parsing);
                    cout << "\t" << input.substr(keep, input.size() - keep) << "\t" << molil
    << "-->" << action << endl;
                    continue;
                }
                parsing.pop();
                for (int j = action.size() - 1; j >= 0; --j)
                {
                    parsing.push(action[j]);
                }
            }
            print_stack(parsing);
            cout << "\t" << input.substr(keep, input.size() - keep) << "\t" << molil << "-->"
    << action << endl;
        }
    }
}
```

Code for LL parser

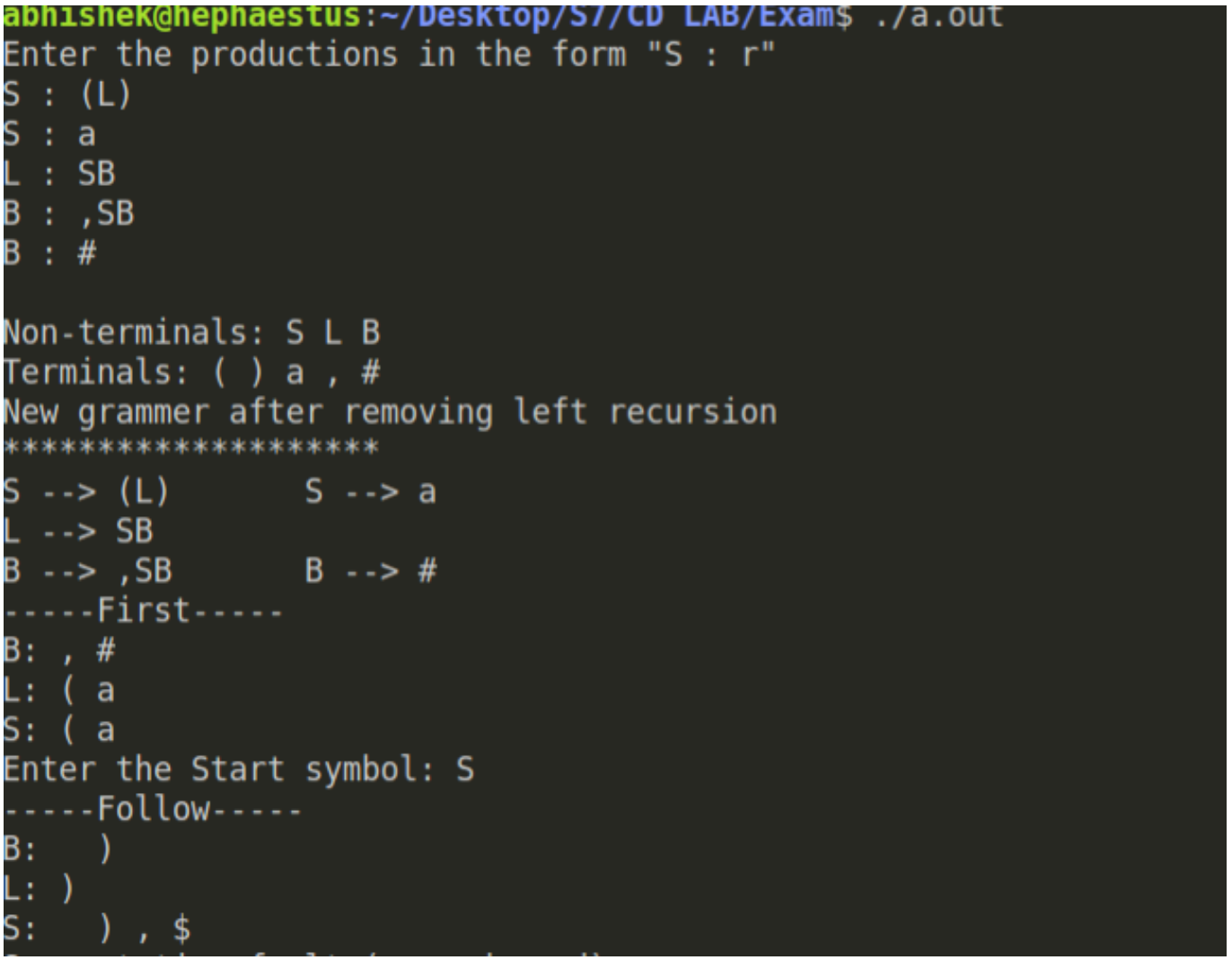
1.5 Output

```
abhishek@hephaestus:~/Desktop/S7/CD LAB/Exam$ ./a.out
Enter the productions in the form "S : r"
S : aBa
B : #
B : bB

Non-terminals: S B
Terminals: a b #
New grammer after removing left recursion
**********
S --> aBa
B --> # B --> bB
-----First-----
B: # b
S: a
Enter the Start symbol: S
-----Follow-----
B: a
S: $
Parsing table
        $       b       a
S                       aBa
B               bB      #
Enter the string to be parsed
abba
stack   Input   output
$S      abba$
$aBa    abba$   S-->aBa
$aB     bba$
$aBb    bba$    B-->bB
$aB     ba$
$aBb    ba$     B-->bB
$aB     a$
$a      a$      B-->#
$       $
parsing finished successfully
abhishek@hephaestus:~/Desktop/S7/CD LAB/Exam$
```

```
Enter the productions in the form "S : r"
E : E+T
E : T
T : T*F
T : F
F : (E)
F : i

Non-terminals: E T F
Terminals: + * i ( ) #
left recursion found in E+T
alpha is +T
first production push back
Extra prodcution pushing back
push back ended
left recursion found in T*F
alpha is *F
first production push back
Extra prodcution pushing back
push back ended
New grammer after removing left recursion
**********
E --> TZ
T --> FY
F --> (E)       F --> i
Z --> +TZ       Z --> #
Y --> *FY       Y --> #
-----First-----
Z: + #
Y: * #
F: ( i
T: ( i
E: ( i
Enter the Start symbol: E
-----Follow-----
Z: $ )
Y: + ) $
F: + ) $ *
T: $ ) +
```




```
abhishek@hephaestus:~/Desktop/S7/CD LAB/Exam$ ./a.out
Enter the productions in the form "S : r"
S : aBa
B : #
B : bB

Non-terminals: S B
Terminals: a b #
New grammer after removing left recursion
**********
S --> aBa
B --> # B --> bB
-----First-----
B: # b
S: a
Enter the Start symbol: S
-----Follow-----
B: a
S: $
Parsing table
        $       b       a
S                       aBa
B               bB      #
Enter the string to be parsed
abba
stack   Input   output
$S      abba$
```

```
$aBa    abba$   S-->aBa
$aB     bba$
$aBb    bba$    B-->bB
$aB     ba$
$aBb    ba$     B-->bB
$aB     a$
$a      a$      B-->#
$       $
parsing finished successfully
```

1.6 Result

Implemented the program to construct a LL(1) parser. It was compiled using g++ version 9.3.0, and executed in Ubuntu 20.04 and the above output was obtained.