

LAB REPORT

COMPILER DESIGN LAB EXPERIMENTS

Amrith M
Roll No. 10
S7 CSE

Contents

1	Cycle -1 (Automata Concepts)	3
1.1	Overview	3
1.2	Program 1	4
1.3	Program 2	4
1.4	Program 3	5
1.5	Program 4	6
1.6	Program which implements Algorithms 1-4	7
2	Cycle -2 (LEX & YACC)	27
2.1	Lex Tool	27
2.2	YACC	30
2.3	Program 1	30
2.4	Program 2	36
2.5	Program 3.1	39
2.6	Program 3.2	41
2.7	Program 3.3	42
3	Cycle -3 (Parsers)	46
3.1	Operator Precedence Parsing	46
3.2	Recursive Descent Parser	46
3.3	First and Follow	47
3.4	Shift Reduce Parser	48
3.5	Program 1	49
3.6	Program 2	53
3.7	Program 3	59
3.8	Program 4	62
4	Cycle -4 (Compiler & Interpreter)	67
4.1	Using Lex and YACC to build a Compiler	67
4.2	Program 1,2,3	68

4.3	Symbol Table	70
4.4	Code Genenration	71
4.5	Program 4	85

1 Cycle -1 (Automata Concepts)

1.1 Overview

In the theory of computation, a branch of theoretical computer science, a deterministic finite automaton (DFA) also known as a deterministic finite acceptor (DFA) and a deterministic finite state machine (DFSM) or a deterministic finite state automaton (DFSFA) is a finite-state machine that accepts or rejects strings of symbols and only produces a unique computation (or run) of the automaton for each input string. Deterministic refers to the uniqueness of the computation. In search of the simplest models to capture finite-state machines, Warren McCulloch and Walter Pitts were among the first researchers to introduce a concept similar to finite automata in 1943.

Formally, A deterministic finite automata (DFA) is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where each of them represents:-

- a finite set of states Q
- a finite set of input symbols called the alphabet Σ
- a transition table δ consisting of a single state as entry for each state-symbol pair
- a start state q_0
- a set of final states F

Similarly, An NFA (Non-Deterministic Finite Automata) without ϵ is also represented by a 5-tuple where the transition table contains an element of the power set of Q as an entry for each state-symbol pair. An NFA with ϵ will also be represented by a 5-tuple where the transition table will contain an element of the power set of Q as an entry for each state-symbol pair as well as state- pair.

Any language that is represented by an NFA with ϵ transitions can be represented by an NFA without ϵ transitions and any language that can be represented by an NFA without ϵ transitions can be represented by a Deterministic Finite Automata (DFA). An NFA without ϵ transitions is a more restricted version of an NFA with ϵ transitions and a DFA is a more restricted version of an NFA without ϵ transitions.

1.2 Program 1

Aim : Write a program to find ϵ – closure of all states of any given *NFA* with ϵ -transitions

Algorithm 1 Given the object containing transition table of an ϵ – *NFA* and a state k , return the ϵ – Closure of state k of the ϵ – *NFA*

function EPSILONCLOSURE(*enfa*, k)

- Initialize a list t containing only state k
- Initialize an iterator to the first element of the list t
- While iterator has not crossed the last element of the list t
 - Append all states in the i – pair in the transition table of *enfa* which is not previously present in list t to t
 - Set the iterator to the next element of the list t
- Return list t as the ϵ – Closure for state k in ϵ – *NFA enfa*

end function

1.3 Program 2

Aim : Write program to convert *NFA* with ϵ transition to *NFA* without ϵ transitions.

Algorithm 2 Given the object *enfa* containing transition table of an ϵ – *NFA*, convert this *enfa* to an *NFA* without ϵ transitions.

function CONVERTTONFA(*enfa*)

- Initialize an empty object of type *NFA* with variable name t
- Initialize $t.numstates = enfa.numstates$, $t.numalphabets = enfa.numalphabets$ and $t.finalstates = enfa.finalstates$
- Iterate through each of the state i in Q
 - Initialize l to the closure of state i of ϵ – *NFA enfa*
 - Iterate through each of the input symbol j in Σ
 - * Initialize an empty list of states f
 - * Iterate through each state k in l
 - Add all states of $enfa.transitiontable[k][j + 1]$ to f
 - * Remove all the duplicates from f
 - * Compute the ϵ – closure c of f
 - * Set $t.transitiontable[i][j] = c$
- Return t as the *NFA* without ϵ -transitions corresponding to the ϵ -*NFA enfa*

end function

1.4 Program 3

Aim : Write program to convert NFA to DFA.

Algorithm 3 Given the object *nfa* containing transition table of an *NFA*, convert this *nfa* to a *DFA* using lazy construction.

function CONVERTTODFA(*nfa*)

- Initialize an empty object of type *DFA* with variable name *dfa*
- Initialize *dfa.num_alphabets* = *nfa.num_alphabets*, *i* = 0
- Initialize a set *lazySet* which stores subsets of *Q* and store {0} in it.
- Create a new row of size *dfa.num_alphabets* and insert into *dfa.table* and initialize all values to -1 .
- While $i < \text{lazySet.size}()$
 - Iterate through each of the input symbol *j* in Σ
 - * Initialize an empty set of states *reachable* and a variable *next* = -1
 - * Iterate *it* through each element in *lazySet*[*i*] and push into *reachable* the set *nfa.table*[*it*][*j*]
 - * Check if *reachable* is already in *lazySet*. If yes, the get the value of *next* from *lazySet*. If not , then
 - Insert into *lazySet*, the set *reachable* and set *next* = *lazySet.size()*
 - Insert *next* into *dfa.finalStates* if any element in *reachable* is a final state of the original *nfa*
 - Create a new row of size *dfa.num_alphabets* and insert into *dfa.table* and initialize all values to -1 .
 - *dfa.table*[*i*][*j*] = *next*
 - Increment *i*
- Return *dfa* as the DFA.

end function

1.5 Program 4

Aim : Write a program to minimize any given DFA.

Algorithm 4 Given the object *dfa* containing transition table of a DFA, convert this *dfa* to minimized DFA.

function MINIMIZEDFA(*dfa*)

- Initialize an empty object of type *dfa* with variable name *minDfa*
- Initialize *minDfa.num_alphabets* = *dfa.num_alphabets*
- Initialize a matrix *m* of size *a.num_states* × *a.num_states* and set every cell in the matrix to 0
- Initialize a flag variable *f* to 1
- For all state pairs (*x*, *y*), Set *m*[*x*][*y*] = 1 if *x* is a final state and *y* is a non-final state or vice-versa (Choose either upper or lower triangle of the matrix).
- While *f*! = 0
 - Set *f* to 0
 - For all states *i* from 0 to *dfa.num_states*
 - * For all states *j* from *i* + 1 to *dfa.num_states*
 - If for any symbol *u* in Σ , *m*[*i*][*j*] = 0 and *m*[*dfa.transitiontable*[*i*][*u*]][*dfa.transitiontable*[*j*][*u*]] = 1, Then Set *m*[*i*][*j*] = 1 and *f* = 1
- Represent those pair of states (*a*, *b*) which has *m*[*a*][*b*] = 0 by a single state *a* in the minimized DFA *minDfa*.
- Return *minDfa* as the minimised DFA.

end function

1.6 Program which implements Algorithms 1-4

The below c++ program takes as input, an ϵ - *NFA* as input and finally outputs a minimised DFA as the final output. Intermediate outputs contain epsilon closure of all states of the ϵ - *NFA*, the corresponding NFA, its DFA and the minimised DFA. To run this program in terminal use "g++ prog.cpp -std=c++11"

```

/****
 *
 * Author : Amrith M
 * Problem : Problems 1-4 In Cycle 1
 * Description : Converts an epsilon-nfa to a minimised dfa
 *
 * Class ENFA represents objects of both epsilon-nfa and nfa
 * Class DFA represents objects of both dfa and minimised dfa
 *
 * Closure is found using Breadth First Search
 * After computing closure of every state, they're stored in
 * → closures vector
 *
 *
 *
 * */

#include <bits/stdc++.h>
using namespace std;

//Must be declared First
class DFA
{
    public:
        int num_states, num_alphabets;
        vector< vector<int> > table;
        set<int> finalStates;

        DFA()
        {
            //Default Constructor
        }

        DFA(int n_s, int n_a)
        {
            num_states = n_s;

```



```
    num_alphabets = n_a;
    table = vector< vector<int> >(n_s,
        ↪ vector<int>(n_a));
}

void printAutomaton()
{

    printf("\nStates : %d, Alphabet : %d
    ↪ \n", num_states, num_alphabets-1);

    for(int i = 0; i < num_states; i++)
    {
        for(int j = 1; j < num_alphabets; j++)
        {
            printf("\ndelta(%d, %d): %d
            ↪ ", i+1, j, table[i][j]+1);
        }
    }

    printf("\nFinal States : { ");

    for(auto it : finalStates)
    {
        cout << it+1 << " ";
    }
    cout << "} \n";
}

/**
 *
 * Minimise DFA using Myhill-Nerode Theorem
 *
 * */

int find(vector<int> &parent, int x, set<int> &path)
{
    path.insert(x);
    if(x != parent[x])
    {
        return find(parent, parent[x], path);
    }
    else
```

```
    {
        return x;
    }
}

int find(int x, vector<int> &parent)
{
    if(x == parent[x])
    {
        return x;
    }
    else
    {
        parent[x] = find(parent[x], parent);
        return parent[x];
    }
}

DFA minimizedDFA()
{
    DFA minDfa(0, num_alphabets);
    vector< vector<int> > m(num_states, vector<int>
        ↪ (num_states, 0));

    bool f = 1;
    for(int i = 0; i < num_states; i++)
    {
        for(int j = i + 1 ; j < num_states; j++)
        {
            if(finalStates.find(i) !=
                ↪ finalStates.end() &&
                ↪ finalStates.find(j) ==
                ↪ finalStates.end())
            {
                m[i][j] = 1;
            }

            if(finalStates.find(j) !=
                ↪ finalStates.end() &&
                ↪ finalStates.find(i) ==
                ↪ finalStates.end())
            {
                m[i][j] = 1;
            }
        }
    }
}
```

```

    }
    }
}

cout << "\nM-N Table(Initial) : \n";
for(int i = 0; i < num_states; i++)
{
    for(int j = i + 1 ; j < num_states; j++)
        cout << m[i][j] << " ";
    cout << endl;
}

while(f)
{
    f = 0;

    for(int i = 0; i < num_states; i++)
    {
        for(int j = i + 1; j < num_states; j++)
        {
            for(int u = 1; u < num_alphabets; u++)
            {
                int i_u = table[i][u];
                int j_u = table[j][u];

                if(i_u > j_u)
                    swap(i_u, j_u);

                if(m[i][j] == 0 && m[i_u][j_u] ==
↪ 1)
                {
                    m[i][j] = 1;
                    f = 1;
                    break;
                }
            }
        }
    }
}

cout << "\nM-N Table : \n";
for(int i = 0; i < num_states; i++)

```

```
{
    for(int j = i + 1 ; j < num_states; j++)
        cout << m[i][j] << " ";
    cout << endl;
}

set<int> visited, unvisited;
set<int> fs;

vector<int> parent(num_states);
for(int i = 0; i < num_states; i++)
{
    parent[i] = i;
}

for(int i = 0; i < num_states; i++)
{
    for(int j = i+1 ; j < num_states; j++)
    {
        if(m[i][j] == 0)
        {
            parent[j] = i;
        }
    }
}

map<int, set<int> > mappings;

for(int i = 0; i < num_states; i++)
{
    set<int> path;
    int x = find(parent,i,path);
    mappings[x] = path;
}

cout << "\nEquivalent States : \n";
for(auto it : mappings)
{
    cout << it.first << " : ";
    for(auto it1 : it.second)
        cout << it1 << " ";
    cout << endl;
}
```

```
minDfa.num_states = mappings.size();
minDfa.table =
    ↪ vector<vector<int>>(num_states,vector<int>(num_alphabets))

map<int,int> numberMap;
int index = 0;

for(auto it : mappings)
{
    numberMap[it.first] = index++;
}

for(auto it : mappings)
{
    for(int j = 1; j < num_alphabets; j++)
    {
        int start = *(it.second.begin());
        int end = table[start][j];
        int parent_element = find(end,parent);

        minDfa.table[numberMap[it.first]][j] =
            ↪ numberMap[parent_element];
    }
}

for(auto it : finalStates)
{
    ↪ minDfa.finalStates.insert(numberMap[find(it,parent)]);
}

return minDfa;
}

};

class ENFA
{
public:
    int num_states, num_alphabets;
    set<int> finalStates;
    vector< vector< vector<int> > > table;
```

```

vector< set<int> > closures;

ENFA(int n_s, int n_a)
{
    num_states = n_s;
    num_alphabets = n_a;
    table = vector< vector< vector<int> >
    ↪ >(n_s,vector< vector<int>
    ↪ >(n_a,vector<int>(0))) );
}

/**
 * Reads only transitions that exist and has the
↪ format :
 * state symbol transitions
 * Eg : 0 1 1 2 3 => delta(0,1) = { 1 2 3}
 *
 * */

void get_automaton()
{
    while(true)
    {
        string s;
        getline(cin,s);
        if(s[0] == 'x')
            break;

        stringstream ss(s);
        vector<int> arr;

        while(ss)
        {
            string temp;
            if(getline(ss,temp, ' '))
            {
                arr.push_back(stoi(temp));
            }
            else
                break;
        }

        for(int i = 2; i < arr.size(); i++)

```

```

        {
            (table[arr[0]][arr[1]]).push_back(arr[i]);
        }
    }

    int x;
    cin >> x;
    for(int i = 0; i < x; i++)
    {
        int a;
        cin >> a;
        finalStates.insert(a);
    }
}

/**
 * Prints the transition function of the automaton
 *
 *
 * */

void printAutomaton()
{
    printf("\nStates : %d, Alphabet : %d\n", num_states, num_alphabets-1);
    for(int i = 0; i < num_states; i++)
    {
        for(int j = 0; j < num_alphabets; j++)
        {
            printf("\ndelta(%d,%d): { ", i, j);
            for(int k = 0; k < table[i][j].size(); k++)
            {
                cout << table[i][j][k] << " ";
            }
            cout << "}" << endl;
        }
    }
    printf("\nFinal States : { ");
    for(auto it : finalStates)
    {
        cout << it << " ";
    }
    cout << "}" << endl;
}

/**

```

```

*
* Computes Closure using Breadth First Search
*
*
* */

set<int> compute_closure(int k)
{
    set<int> closureSet;
    queue<int> pendingNodes;
    vector<bool> visited(num_states, false);

    pendingNodes.push(k);
    visited[k] = true;

    while(!pendingNodes.empty())
    {
        int node = pendingNodes.front();
        pendingNodes.pop();

        closureSet.insert(node);

        for(int i = 0; i < (table[node][0]).size();
            ↪ i++)
        {
            int x = table[node][0][i];
            if(!visited[x])
            {
                pendingNodes.push(x);
                closureSet.insert(x);
            }
        }
    }

    return closureSet;
}

/****
*
* Converts ENFA to NFA using the standard
↪ algorithm. Updates final states also.
* returns and object of ENFA which represents the
↪ requied nfa.

```



```
*
* */

ENFA convert_to_nfa()
{
    ENFA nfa(num_states, num_alphabets);
    nfa.finalStates = finalStates;

    for(int i = 0; i < num_states; i++)
    {
        set<int> closure = closures[i];
        for(int j = 1; j < num_alphabets; j++)
        {
            vector<int> states;
            for(auto it : closure)
            {
                ↪ states.insert(states.end(), table[it][j].begin()
            }

            set<int>
            ↪ stateSet(states.begin(), states.end());
            set<int> resultant;

            //Computing e-closure of this set w.r.t
            ↪ ENFA this
            for(auto it : stateSet)
            {
                ↪ resultant.insert(closures[it].begin(), closures

            }

            vector<int>
            ↪ finalAns(resultant.begin(), resultant.end());

            nfa.table[i][j] = finalAns;
        }
    }

    for(auto it : finalStates)
    {
        for(int i = 0; i < num_states; i++)
        {
```

```
        if(closures[i].find(it) !=
            ↪ closures[i].end())
        {
            nfa.finalStates.insert(i);
        }
    }
    return nfa;
}

/**
 * Gets a set of states corresponing to a dfa state
 *
 * */

set<int> nfa_state(int st)
{
    st++;
    set<int> states;
    int i = 31;

    while(i >= 0)
    {
        if(st & (1 << i))
            states.insert(i);
        i--;
    }
    return states;
}

/**
 * Gets an integer that represents a set of states of
↪ an nfa
 * */

int dfa_state(set<int> states)
{
    int reqState = 0;
    for(auto it : states)
    {
        reqState += (int)pow(2, it);
    }
}
```

```

        return reqState - 1;
    }

    /**
     *
     * Only objects of NFA format is allowed to invoke
    ↪ this function
     * SUBSET CONSTRUCTION  $O(n \cdot 2^n)$ 
     *
     * */
DFA convert_to_dfa()
{
    int dfa_states = (int)pow(2, num_states);
    DFA dfa(dfa_states, num_alphabets);

    for(int i = 0; i < dfa_states-1; i++)
    {
        set<int> states_in_nfa = nfa_state(i);

        for(int j = 1; j < num_alphabets; j++)
        {
            vector<int> states;
            for(auto it : states_in_nfa)
            {
                ↪ states.insert(states.end(), table[it][j].begin()
            }

            set<int> f(states.begin(), states.end());
            if(f.empty())
                dfa.table[i][j] = dfa_states - 1;
            else
            {
                dfa.table[i][j] = dfa_state(f);
            }
        }
    }

    //Dead State Config
    for(int j = 1; j < num_alphabets; j++)

```

```

        dfa.table[(1 << num_states) - 1][j] = (1 <<
        ↪ num_states) - 1;

    /**
     * Generate Final states
     * Get all subsets that contain any of the final
    ↪ states of the nfa
     *
     * */

    set<int> actualFinal;
    for(int i = 0; i < dfa_states-1; i++)
    {
        set<int> fs = nfa_state(i);
        for(auto it : finalStates)
        {
            if(fs.find(it) != fs.end())
            {
                actualFinal.insert(i);
            }
        }
    }

    dfa.finalStates = actualFinal;

    //dfa.printAutomaton();

    /**
     * Doing BFS To optimse state Count
     * Collecting all states that has a path from the
    ↪ start state and mapping them from 0 to size
     *
     * */

    queue<int> q;
    vector<bool> visited(dfa_states, 0);

    q.push(0);
    visited[0] = 1;

    vector< int > req_states;
    int new_state_count = 0;

```

```
while(!q.empty())
{
    int curr = q.front();
    q.pop();

    req_states.push_back(curr);

    new_state_count++;

    for(int i = 1; i < num_alphabets; i++)
    {
        int next = dfa.table[curr][i];
        if(!visited[next])
        {
            visited[next] = 1;
            q.push(next);
        }
    }
}

printf("\nNumber of States in BFS Optimised DFA is
↪ : %d \n", new_state_count);

//Mapping Arbitrary states to 0 - n

unordered_map<int, int> mappings;
int index = 0;
for(auto it : req_states)
{
    if(mappings.find(it) == mappings.end())
    {
        mappings[it] = index++;
    }
}

DFA optimised(index, num_alphabets);

for(auto it : req_states)
{
    for(int i = 1; i < num_alphabets; i++)
    {
```

```

        optimised.table[mappings[it]][i] =
        ↪ mappings[dfa.table[it][i]];
    }
}

for(auto it : dfa.finalStates)
{
    if(mappings.find(it) != mappings.end())

        ↪ (optimised.finalStates).insert(mappings[it]);
}

/**
 *
 * Print Without Mapping. (states will get weird
↪ numbers)
 *
 * */

/*

for(auto it : req_states)
{
    for(int i = 1; i < num_alphabets; i++)
    {
        printf("\ndelta(%d,%d): %d
↪ ",it,i,dfa.table[it][i]);
    }
}
cout << "\nFinal : ";
for(auto it : req_states)
    if(dfa.finalStates.find(it) !=
↪ dfa.finalStates.end())
        cout << it << " ";

cout << endl;

*/

return optimised;
}

```

```
/**
 *
 * Converts NFA to DFA using Lazy Construction
 * Creates a new state only if there is a path to it
 *
 */

DFA convert_to_dfa_lazy()
{
    DFA dfa(0, num_alphabets);

    vector < set<int> > lazySet;

    int i = 0;

    lazySet.push_back({0});

    ↪ dfa.table.push_back(vector<int>(num_alphabets,-1));

    if(finalStates.find(0) != finalStates.end())
        dfa.finalStates.insert(0);

    while(i < lazySet.size())
    {
        for(int j = 1; j < num_alphabets; j++)
        {
            int new_state = -1;

            set<int> reachable;
            for(auto it : lazySet[i])
            {
                ↪ reachable.insert(table[it][j].begin(),table[it][j].end());
            }

            for(int x = 0; x < lazySet.size(); x++)
            {
                if(reachable == lazySet[x])
                {
                    new_state = x;
                    break;
                }
            }
        }
        lazySet.push_back(reachable);
        i++;
    }
}
```

```
        }
    }

    if(new_state == -1)
    {
        new_state = lazySet.size();
        lazySet.push_back(reachable);

        for(auto it : reachable)
        {
            if(finalStates.find(it) !=
               ↪ finalStates.end())
            {

                ↪ dfa.finalStates.insert(new_state);
                break;
            }
        }

        ↪ dfa.table.push_back(vector<int>(num_alphabets,

    }

    dfa.table[i][j] = new_state;
}

i++;
}

dfa.num_states = dfa.table.size();
return dfa;
}

};

int main()
{
    int n_s, n_a;
    cin >> n_s >> n_a;
```



```

ENFA enfa(n_s, n_a + 1);
enfa.get_automaton();

cout << "\n\nProblem - 1 : Epsilon Closure of
↪ e-nfa\n===== \n";

for(int i = 0; i < enfa.num_states; i++)
{
    set<int> res = enfa.compute_closure(i);
    printf("\n%d : { ", i);

    for(auto it : res)
        cout << it << " ";
    cout << " }";

    enfa.closures.push_back(res);
}

cout << "\n\nProblem - 2 : Generating NFA
↪ \n===== \n";
ENFA nfa = enfa.convert_to_nfa();
nfa.printAutomaton();

cout << "\n\nProblem - 3 : Generating DFA
↪ \n===== \n";
DFA dfa = nfa.convert_to_dfa_lazy();
dfa.printAutomaton();

cout << "\n\nProblem - 4 : Generating Minimised DFA
↪ \n===== \n";
DFA min_dfa = dfa.minimizeDFA();
min_dfa.printAutomaton();

return 0;
}

```

Input & Output

The following 5 images contains the input and outputs of the 4 problems given.

```
Amriths-Air:Cycle-1-Automata amrithm98$ g++ enfa_to_min_dfa.cpp --std=c++11
Amriths-Air:Cycle-1-Automata amrithm98$ ./a.out
21 2
0 0 1 7
1 0 2 4
2 1 3
4 2 5
3 0 6
5 0 6
6 0 1 7
7 1 8
8 0 9 11
9 1 10
11 2 12
10 0 13
12 0 13
13 0 14
14 0 15 17
15 1 16
17 2 18
16 0 19
18 0 19
19 0 20
x
1
20
```

Problem - 1 : Epsilon Closure of e-nfa

```
=====
0 : { 0 1 2 4 7 }
1 : { 1 2 4 }
2 : { 2 }
3 : { 1 2 3 4 6 7 }
4 : { 4 }
5 : { 1 2 4 5 6 7 }
6 : { 1 2 4 6 7 }
7 : { 7 }
8 : { 8 9 11 }
9 : { 9 }
10 : { 10 13 14 15 17 }
11 : { 11 }
12 : { 12 13 14 15 17 }
13 : { 13 14 15 17 }
14 : { 14 15 17 }
15 : { 15 }
16 : { 16 19 20 }
17 : { 17 }
18 : { 18 19 20 }
19 : { 19 20 }
20 : { 20 }
```

```
Problem - 2 : Generating NFA
=====

States : 21, Alphabet : 2

delta(0,1): { 1 2 3 4 6 7 8 9 11 }
delta(0,2): { 1 2 4 5 6 7 }
delta(1,1): { 1 2 3 4 6 7 }
delta(1,2): { 1 2 4 5 6 7 }
delta(2,1): { 1 2 3 4 6 7 }
delta(3,1): { 1 2 3 4 6 7 8 9 11 }
delta(3,2): { 1 2 4 5 6 7 }
delta(4,2): { 1 2 4 5 6 7 }
delta(5,1): { 1 2 3 4 6 7 8 9 11 }
delta(5,2): { 1 2 4 5 6 7 }
delta(6,1): { 1 2 3 4 6 7 8 9 11 }
delta(6,2): { 1 2 4 5 6 7 }
delta(7,1): { 8 9 11 }
delta(8,1): { 10 13 14 15 17 }
delta(8,2): { 12 13 14 15 17 }
delta(9,1): { 10 13 14 15 17 }
delta(10,1): { 16 19 20 }
delta(10,2): { 18 19 20 }
delta(11,2): { 12 13 14 15 17 }
delta(12,1): { 16 19 20 }
delta(12,2): { 18 19 20 }
delta(13,1): { 16 19 20 }
delta(13,2): { 18 19 20 }
delta(14,1): { 16 19 20 }
delta(14,2): { 18 19 20 }
delta(15,1): { 16 19 20 }
delta(17,2): { 18 19 20 }
Final States : { 16 18 19 20 }
```

```
Problem - 3 : Generating DFA
=====

States : 9, Alphabet : 2

delta(1, 1): 2
delta(1, 2): 3
delta(2, 1): 4
delta(2, 2): 5
delta(3, 1): 2
delta(3, 2): 3
delta(4, 1): 6
delta(4, 2): 7
delta(5, 1): 8
delta(5, 2): 9
delta(6, 1): 6
delta(6, 2): 7
delta(7, 1): 8
delta(7, 2): 9
delta(8, 1): 4
delta(8, 2): 5
delta(9, 1): 2
delta(9, 2): 3
Final States : { 6 7 8 9 }
```

```
States : 8, Alphabet : 2

delta(1, 1): 2
delta(1, 2): 1
delta(2, 1): 3
delta(2, 2): 4
delta(3, 1): 5
delta(3, 2): 6
delta(4, 1): 7
delta(4, 2): 8
delta(5, 1): 5
delta(5, 2): 6
delta(6, 1): 7
delta(6, 2): 8
delta(7, 1): 3
delta(7, 2): 4
delta(8, 1): 2
delta(8, 2): 1
Final States : { 5 6 7 8 }
```

2 Cycle -2 (LEX & YACC)

2.1 Lex Tool

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed. The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

The general format of Lex source is:

definitions rules user subroutines

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

%%

(no definitions, no rules) which translates into a program which copies the input to the output unchanged. In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions (see section 3) and the right column contains actions, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

integer printf("found keyword INT");

to look for the string integer in the input stream and print the message “found keyword INT” whenever it appears. In this example the host procedural language is C and the C library function `printf` is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour printf("color");
mechanise printf("mechanize");
petrol printf("gas");
```

would be a start. These rules are not quite enough, since the word petroleum would become gaseum;

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
\$	end of line
a b	a or b
(ab) +	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Table 1: Pattern Matching Primitives

Expression	Matches
<code>abc</code>	<code>abc</code>
<code>abc*</code>	<code>ab abc abcc abccc ...</code>
<code>abc+</code>	<code>abc abcc abccc ...</code>
<code>a(bc)+</code>	<code>abc abcbc abcbcbc ...</code>
<code>a(bc)?</code>	<code>a abc</code>
<code>[abc]</code>	one of: <code>a</code> , <code>b</code> , <code>c</code>
<code>[a-z]</code>	any letter, <code>a-z</code>
<code>[a\ -z]</code>	one of: <code>a</code> , <code>-</code> , <code>z</code>
<code>[-az]</code>	one of: <code>-</code> , <code>a</code> , <code>z</code>
<code>[A-Za-z0-9]+</code>	one or more alphanumeric characters
<code>[\t\n]+</code>	whitespace
<code>[^ab]</code>	anything except: <code>a</code> , <code>b</code>
<code>[a^b]</code>	one of: <code>a</code> , <code>^</code> , <code>b</code>
<code>[a b]</code>	one of: <code>a</code> , <code> </code> , <code>b</code>
<code>a b</code>	one of: <code>a</code> , <code>b</code>

Table 2: Pattern Matching Examples

Name	Function
<code>int yylex(void)</code>	call to invoke lexer, returns token
<code>char *yytext</code>	pointer to matched string
<code>yylen</code>	length of matched string
<code>yyval</code>	value associated with token
<code>int yywrap(void)</code>	wrapup, return 1 if done, 0 if not done
<code>FILE *yyout</code>	output file
<code>FILE *yyin</code>	input file
<code>INITIAL</code>	initial start condition
<code>BEGIN</code>	condition switch start condition
<code>ECHO</code>	write matched string

Table 3: Lex Predefined Variables

2.2 YACC

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express context-free languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

$$\begin{aligned}E &\rightarrow E + E \\E &\rightarrow E * E \\E &\rightarrow \text{id}\end{aligned}$$

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E (expression) are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

$$\begin{aligned}E &\rightarrow E * E \text{ (r2)} \\&\rightarrow E * z \text{ (r3)} \\&\rightarrow E + E * z \text{ (r1)} \\&\rightarrow E + y * z \text{ (r3)} \\&\rightarrow x + y * z \text{ (r3)}\end{aligned}$$

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to reduce an expression to a single nonterminal. This is known as bottom-up or shift-reduce parsing and uses a stack for storing terms.

Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

2.3 Program 1

Aim : Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new line.

In this program, we are using a custom language which is displayed below and we are identifying each token in the program using regular expressions.

```

prog
integer a,b
begin
    read n;

    if a < 10
    then
        b := 1;
    else;
    endif;

    while a < 10
    do
        b := 5*a;
        a:=a+1;
    endwhile;

    write a;
    write b;
end

```

```

/**
 * Author : Amrith M
 * This code tokenizes the given program and prints what each
 * token is
 * This code is written in c++ and uses regex library
 * regex_match is used to match regexp with string
 *
 *
 * */
#include <bits/stdc++.h>
using namespace std;

set<string> keywords = {"prog" , "begin" , "end" , "if" ,
    → "then" , "else" , "endif" , "while" , "do" , "endwhile" ,
    → "read" , "write", "integer" };
set<string> operators = { "!=" , "+" , "-" , "*" , "/" , "=" ,
    → "<" , ">" };
set<char> symbols = { '.', ',', '(', ')' };
set<char> ops = { ':', '+', '-', '*', '/', '<', '>', ';', ',', ' ' };

regex identifier("[a-zA-Z_][_a-zA-Z0-9]*");
regex digits("[1-9][0-9]*");

vector< pair<string,string> > tokens;

```



```
vector<string> splitDelimiter(string s, char ch)
{
    stringstream ss(s);
    vector<string> arr;

    while(ss)
    {
        string temp;
        if(getline(ss,temp,ch))
        {
            arr.push_back(temp);
        }
        else
            break;
    }

    return arr;
}

bool putMatching(string current)
{
    if(keywords.find(current) != keywords.end())
    {
        tokens.push_back({"keyword",current});
        return true;
    }

    if(operators.find(current) != operators.end())
    {
        tokens.push_back({"operator",current});
        return true;
    }

    if(regex_match(current,identifier))
    {
        tokens.push_back({"identifier",current});
        return true;
    }

    if(regex_match(current,digits))
    {
        tokens.push_back({"literal",current});
    }
}
```

```
        return true;
    }

    return false;
}

string stripStr(string stripString)
{
    while(!stripString.empty() &&
        ↪ std::isspace(*stripString.begin()))
        stripString.erase(stripString.begin());

    while(!stripString.empty() &&
        ↪ std::isspace(*stripString.rbegin()))
        stripString.erase(stripString.length()-1);

    return stripString;
}

void parse(string s)
{
    vector<string> spaceSplit = splitDelimiter(s, ' ');
    vector<string> strings;

    for(auto it : spaceSplit)
    {
        vector<string> temp = splitDelimiter(it, ';');
        strings.insert(strings.end(), temp.begin(), temp.end());
    }

    for(int i = 0; i < strings.size(); i++)
    {
        string current = strings[i];
        current = stripStr(current);

        if(!putMatching(current))
        {
            int left = 0, right = 0;
            string temp = "";
            while(right < current.size() && left <= right)
            {
                if(symbols.find(current[right]) != symbols.end())
                {

```

```
        putMatching(temp);
        temp = "";
        tokens.push_back({"symbol", string(1, current[right])});
        putMatching(current.substr(left, right-left+1));
        left = right+1;
        right++;
        continue;
    }
    else if (ops.find(current[right]) != ops.end())
    {
        putMatching(temp);
        temp = "";
        if (current[right] == ':')
        {
            tokens.push_back({"operator", "!="});
            putMatching(current.substr(left, right-left));
            left = right+2;
            right += 2;
        }
        else
        {
            tokens.push_back({"operator", string(1, current[right])});
            putMatching(current.substr(left, right-left));
            left = right+1;
            right++;
        }
        continue;
    }
    else
    {
        temp += current[right];
        right++;
    }
}
if (right >= left)
{
    putMatching(current.substr(left, right-left));
}
}
}
}
int main()
{
```

```
ifstream in("input.txt");
ofstream op("output.txt");

string s;
while(getline(in,s))
{
    parse(s);
}

for(auto it : tokens)
{
    cout << setw(12) << it.first << " : " << it.second <<
    ↪ endl;
}

return 0;
}
```

Input & Output

```
prog
integer a,b.
begin
    read a;
    if a < 10
    then
        b := 1;
    else
        endif;

    while a < 10
    do
        b := 5*a;
        a := a+1;
    endwhile;

    write a;
    write b;
end
```

```

keyword : prog
keyword : integer
identifier : a
symbol : ,
identifier : b
keyword : begin
keyword : read
identifier : n
keyword : if
identifier : a
operator : <
literal : 10
keyword : then
identifier : b
operator : :=
literal : 1
keyword : else
keyword : endif
keyword : while
identifier : a
operator : <
literal : 10
keyword : do
identifier : b
operator : :=
literal : 5
operator : *
literal : 5
identifier : a
identifier : a
operator : :=
identifier : a
identifier : a
operator : +
identifier : a
literal : 1
keyword : endwhile
keyword : write
identifier : a
keyword : write
identifier : b
keyword : end

```

2.4 Program 2

Aim : Implementation of Lexical Analyzer using Lex Tool.

In this program, we are tokenizing a custom language using lex tool. We give regular expressions and their corresponding return value.

Lex Code

```

%{
#include "prog.h"
%}

letter [a-zA-Z]
digitt [0-9]
id {letter}*|({letter}{digitt})+
notid ({digitt}{letter})+

%%

"."      return DOT;
":="    return ASSIGN_OP;
";"      return SEMICOLON;

```

```

"+"      return ADD_OP;
-        return SUB_OP;
"*"      return MUL_OP;
"/"      return DIV_OP;
=        return EQU_OP;
"<"      return LT;
">"      return GT;
"<="     return LTE;
">="     return GTE;
[ (, ), {, } ] return LITERAL;
read|
write|
prog|
begin|
integer|
while|
do|
if|
else|
endif|
end|
then|
endwhile return KEYWORD;
{id}      return IDENTIFIER;
{notid}   { printf("\n%s is not an identifier at line %d\n",
→ yytext,yylineno); exit(0);}
[1-9][0-9]* return DIGIT;
[\n]      {yylineno++;}
[ \t]     ;
%%

```

```

int yywrap(void)
{
    return 1;
}

```

Driver Program

```

#include <iostream>
#include "prog.h"

extern int yylex();
extern int yylineno;
extern char* yytext;

```

```
int main(void)
{
    int ntoken;
    ntoken = yylex();
    while(ntoken)
    {
        printf("\nToken : %d Value : %s",ntoken,yytext);
        ntoken = yylex();
    }
    return 0;
}
```

Input & Output

```
prog
integer a,b.
begin
    read a;
    if a < 10
    then
        b := 1;
    else
        endif;

    while a < 10
    do
        b := 5*a;
        a := a+1;
    endwhile;

    write a;
    write b;
end
```

```

Token : 3 Value : prog
Token : 3 Value : integer
Token : 1 Value : a
Token : 15 Value : ,
Token : 1 Value : b
Token : 3 Value : begin
Token : 3 Value : read
Token : 1 Value : readn
Token : 9 Value : ;
Token : 3 Value : if
Token : 1 Value : a
Token : 4 Value : <
Token : 8 Value : 10
Token : 3 Value : then
Token : 1 Value : b
Token : 2 Value : :=
Token : 8 Value : 1
Token : 9 Value : ;
Token : 3 Value : else
Token : 9 Value : ;
Token : 3 Value : endif
Token : 9 Value : ;
Token : 3 Value : while
Token : 1 Value : a
Token : 4 Value : <
Token : 8 Value : 10
Token : 3 Value : do
Token : 1 Value : b
Token : 2 Value : :=
Token : 8 Value : 5
Token : 12 Value : *
Token : 1 Value : a
Token : 9 Value : ;
Token : 1 Value : a
Token : 2 Value : :=
Token : 1 Value : a
Token : 10 Value : +
Token : 8 Value : 1
Token : 9 Value : ;
Token : 3 Value : endwhile
Token : 9 Value : ;
Token : 3 Value : write
Token : 1 Value : a

```

2.5 Program 3.1

*Aim : : Generate YACC Specifications \rightarrow Program to recognize a valid arithmetic expression that uses operator +, , * and /.*

Lex Code

```

%{
    #include "y.tab.h"
%}
%%
[a-zA-Z_] [a-zA-Z0-9] *      {return ID;}
[0-9] +                      {return NUM;}
[\t]                          {;}
[\n]                          {return 0;}
.                             {return yytext[0];}
%%
int yywrap(void)
{
    {return 1;}
}

```


YACC Code

```
%{
    #include<stdio.h>
    #include<stdlib.h>
    int n = 2;
    void yyerror();
}%

%start stmt
%token ID NUM
%left '+' '-'
%left '/' '*'
%%

stmt : expr
      | ID '=' expr
;
expr : expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | '(' expr ')'
      | ID
      | NUM
;
%%

void main()
{
    while(n)
    {
        printf("INPUT AN EXPRESSION : ");
        yyparse();
        printf("VALID EXPRESSION IDENTIFIED\n\n");
    }
}

void yyerror()
{
    printf("EXPRESSION IS INVALID\n\n");
    exit(0);
}
```

Input & Output

```
Amriths-Air:Prog-3-A-Yacc-Arithmetic amrithm98$ ./a.out
INPUT AN EXPRESSION : i+i*i
VALID EXPRESSION IDENTIFIED

INPUT AN EXPRESSION : i**
EXPRESSION IS INVALID
```

2.6 Program 3.2

Aim : *Generate YACC Specifications → Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.*

Lex Code

```
%{
    #include "y.tab.h"
}%
%%
[a-zA-Z_][_a-zA-Z0-9]*      {return ID;}
[\t]                        {;}
[\n]                        {return 0;}
.                            {return yytext[0];}
%%
int yywrap(void)
{
    return 1;
}
```

YACC Code

```
%{
    #include<stdlib.h>
    #include<stdio.h>
    void yyerror();
    int n = 1;
}%

%start stmt
%token ID

%%
stmt : ID
;
%%
void main()
{
```

```

while(n)
{
printf("INPUT AN IDENTIFIER : ");
yyparse();
printf("VALID IDENTIFIER\n\n");
}
}
void yyerror()
{
printf("INVALID IDENTIFIER\n\n");
exit(0);
}

```

Input & Output

```

Amriths-Air:Prog-3-B-Yacc-Identifier amrithm98$ ./a.out
INPUT AN IDENTIFIER : cat_face
VALID IDENTIFIER

INPUT AN IDENTIFIER : 98amr
INVALID IDENTIFIER

```

2.7 Program 3.3

Aim : Generate YACC Specifications → Implementation of Calculator using LEX and YACC.

Lex Code

```

%{
#include <stdio.h>
#include "y.tab.h"
}%

%option noyywrap

%%

"print"      {return print;}
"exit"       {return end;}
[a-zA-Z]     {yylval.id = yytext[0]; return identifier;}
[0-9]+       {yylval.n = atoi(yytext); return num;}
[\t\n]       ;
[-=+ / ; ( )] {return yytext[0];}
"*"          {return yytext[0];}
.            {ECHO; yyerror();}
%%

```

YACC Code

```

%{
    #include<stdio.h>
    #include<stdlib.h>
    void yyerror();
    int sym[52];
    int value(char c);
    void update(char s,int val);
}%

%union{int n; char id;}
%start stmt
%token print end
%token <n> num
%token <id> identifier
%type <n> stmt exp term
%type <idnt> assign
%right '='
%left '+' '-'
%left '*' '/'
%%

stmt : assign ';'
    | end ';'
    | print exp ';'
    ↪ %d\n", $2);}
    | stmt assign ';'
    | stmt print exp ';'
    ↪ %d\n", $3);}
    | stmt end ';'

assign : identifier '=' exp
;

exp : term
    | '(' exp ')'
    | exp '=' exp
    | exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
;

term : identifier

```

```

    {;}
    {exit(EXIT_SUCCESS);}
    {printf("\nValue is :

    {;}
    {printf("\nValue is :

    {exit(EXIT_SUCCESS)};};

    {update($1,$3);}

    {$$ = $1;}
    {$$ = $2;}
    {$$ = $3;}
    {$$ = $1+$3;}
    {$$ = $1-$3;}
    {$$ = $1*$3;}
    {$$ = $1/$3;}

    {$$ = value($1);}

```

```
| num                                {$$ = $1;}
;

%%

int idx(char s)
{
    int i = -1;
    if(islower(s))
    {
        i = s - 'a' + 26;
    }
    else if(isupper(s))
    {
        i = s - 'A';
    }
    return i;
}

int value(char s)
{
    int i = idx(s);
    return sym[i];
}

void update(char s, int val)
{
    int i = idx(s);
    sym[i] = val;
}

int main(void)
{
    int j;

    for(j = 0; j < 52; j++)
        sym[j] = 0;

    return yyparse();
}

void yyerror()
{

```

```
    printf("\nError in Parsing");  
}
```

Input & Output

```
Amriths-Air:Prog-3-C-Yacc-Calc amrithm98$ cat input.txt  
a=5;  
b=10;  
c=a+b;  
d=a-b;  
e=a/b;  
f=a*b;  
g=f+5*3;  
h=a+(a*b*c);  
print a;  
print b;  
print c;  
print d;  
print e;  
print f;  
print g;  
print h;  
print 1+2+3*4;  
end;Amriths-Air:Prog-3-C-Yacc-Calc amrithm98$ ./a.out < input.txt  
  
Value is : 5  
Value is : 10  
Value is : 15  
Value is : -5  
Value is : 0  
Value is : 50  
Value is : 65  
Value is : 755  
Value is : 15
```

3 Cycle -3 (Parsers)

3.1 Operator Precedence Parsing

A grammar that is generated to define the mathematical operators is called operator grammar with some restrictions on grammar. An operator precedence grammar is a context-free grammar that has the property that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its right-hand side.

An operator precedence parser is a one of the bottom-up parser that interprets an operator-precedence grammar. This parser is only used for operator grammars. Ambiguous grammars are not allowed in case of any parser except operator precedence parser. There are two methods for determining what precedence relations should hold between a pair of terminals:

- Use the conventional associativity and precedence of operator.
- The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

This parser relies on the following three precedence relations: \leq , \cong , $>$.

- $a \leq b$ This means a yields precedence to b.
- $a > b$ This means a takes precedence over b.
- $a \cong b$ This means a has precedence as b.

	id	+	*	\$
id		$>$	$>$	$>$
+	$<$		$<$	$>$
*	$<$	$>$		$>$
\$	$<$	$<$	$<$	

3.2 Recursive Descent Parser

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may

or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature. Recursive descent with backtracking is a technique that determines which production to use by trying each production in turn. Recursive descent with backtracking is not limited to LL(k) grammars, but is not guaranteed to terminate unless the grammar is LL(k). Even when they terminate, parsers that use recursive descent with backtracking may require exponential time.

3.3 First and Follow

Each time a predictive parser makes a decision, it needs to determine which production rule to apply to the leftmost non-terminal in an intermediate form, based on the next terminal (i.e. the lookahead symbol). This is the significance of the FIRST sets: they tell you when a nonterminal can produce the lookahead symbol as the beginning of a statement, so that it can be matched away and reduce the input. FOLLOW covers the possibility that the leftmost non-terminal can disappear, so that the lookahead symbol is not actually a part of what we're presently expanding, but rather the beginning of the next construct. This is the significance of the FOLLOW sets: they tell you when a non-terminal can hand you the lookahead symbol at the beginning of a statement by disappearing. Choosing productions that give ϵ doesn't reduce the input string, but you still have to make a rule for when the parser needs to take them, and the appropriate conditions are found from the FOLLOW set of the troublesome non-terminal.

FIRST(X) FOR ALL GRAMMAR SYMBOLS X

Apply following rules:

1. If X is terminal, $\text{FIRST}(X) = \{X\}$.
2. If $X \rightarrow \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.
3. If X is a non-terminal, and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, and ϵ is in all of $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_k)$, then add ϵ to $\text{FIRST}(X)$.
4. If X is a non-terminal, and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then add a to $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$.

Applying rules 1 and 2 is obvious. Applying rules 3 and 4 for $\text{FIRST}(Y_1 Y_2 \dots Y_k)$ can be done as follows:

Add all the non- ϵ symbols of $\text{FIRST}(Y_1)$ to $\text{FIRST}(Y_1 Y_2 \dots Y_k)$. If $\epsilon \in \text{FIRST}(Y_1)$, add all the non- ϵ symbols of $\text{FIRST}(Y_2)$. If $\epsilon \in \text{FIRST}(Y_1)$ and $\epsilon \in \text{FIRST}(Y_2)$, add all the non- ϵ symbols of $\text{FIRST}(Y_3)$, and so on. Finally, add ϵ to $\text{FIRST}(Y_1 Y_2 \dots Y_k)$ if $\epsilon \in \text{FIRST}(Y_i)$, for all $1 \leq i \leq k$.

FOLLOW(A) FOR ALL NON-TERMINALS A

Apply the following rules:

1. If \$ is the input end-marker, and S is the start symbol, $\$ \in \text{FOLLOW}(S)$.
2. If there is a production, $A \rightarrow \alpha B \beta$, then $(\text{FIRST}(\beta) - \epsilon) \subseteq \text{FOLLOW}(B)$.
3. If there is a production, $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where $\epsilon \in \text{FIRST}(\beta)$, then $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$.

Note that unlike the computation of FIRST sets for non-terminals, where the focus is on *what a non-terminal generates*, the computation of FOLLOW sets depends upon *where the non-terminal appears on the RHS of a production*.

3.4 Shift Reduce Parser

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. In other words, it is a process of reducing (opposite of deriving a symbol using a production rule) a string w to the start symbol of a grammar. At every (reduction) step, a particular substring matching the RHS of a production rule is replaced by the symbol on the LHS of the production.

A general form of shift-reduce parsing is LR (scanning from Left to right and using Right-

most derivation in reverse) parsing, which is used in a number of automatic parser generators like Yacc, Bison, etc. A handle of a string is a substring that matches the RHS of a production and whose reduction to the non-terminal (on the LHS of the production) represents one step along the reverse of a rightmost derivation toward reducing to the start symbol. The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. It is always possible to add terminal symbols to the end of a viable prefix to obtain a right-sentential form. Example :

Stack	Input Buffer	Parsing Action
\$	id+id+id\$	Shift
\$id	+id+id\$	Reduce by $S \rightarrow id$
\$S	+id+id\$	Shift
\$S+	id+id\$	Shift
\$S+id	+id\$	Reduce by $S \rightarrow id$
\$S+S	+id\$	Shift
\$S+S+	id\$	Shift
\$S+S+id	\$	Reduce by $S \rightarrow id$
\$S+S+S	\$	Reduce by $S \rightarrow S+S$
\$S+S	\$	Reduce by $S \rightarrow S+S$
\$S	\$	Accept

3.5 Program 1

Aim : : Develop an operator precedence parser for a given language.

```
#include <bits/stdc++.h>
using namespace std;

vector<char> st(100);
vector<string> handles;
```

```
vector<string> precedence(9);
string prevHandle;
int ptr;
int top;

int getOperatorIndex(char value)
{
    switch(value)
    {
        case '+':
            return 0;
        case '-':
            return 1;
        case '*':
            return 2;
        case '/':
            return 3;
        case '^':
            return 4;
        case 'i':
            return 5;
        case '(':
            return 6;
        case ')':
            return 7;
        case '$':
            return 8;
        default:
            return -1;
    }
}

void shift(char c)
{
    st[++top] = c;
    ptr++;
}

bool reduce()
{
    bool found;

    for(int i = 0; i < handles.size(); i++)
```

```

{
    int size = handles[i].size();

    if(st[top] == handles[i][0] && top + 1 >= size )
    {
        found = true;
        int j;
        for(j = 0; j < size; j++)
        {
            if(st[top - j] != handles[i][j])
            {
                found = false;
                break;
            }
        }

        if(found)
        {
            st[top - j + 1] = 'E';
            top = top - j + 1;
            prevHandle = handles[i];
            return true;
        }
    }
}

return false;
}

void setup()
{
    ptr = 0;
    st[0] = '$';
    handles = {"i", "E+E", "E*E", "E-E", "E^E", ")E("};
    top = 0;
    //Precedence Table
    precedence[0] = {'>', '>', '<', '<', '<', '<', '<', '<', '>', '>'};
    precedence[1] = {'>', '>', '<', '<', '<', '<', '<', '<', '>', '>'};
    precedence[2] = {'>', '>', '>', '>', '<', '<', '<', '<', '>', '>'};
    precedence[3] = {'>', '>', '>', '>', '<', '<', '<', '<', '>', '>'};
    precedence[4] = {'>', '>', '>', '>', '<', '<', '<', '<', '>', '>'};
    precedence[5] = {'>', '>', '>', '>', '>', 'e', 'e', '>', '>'};
    precedence[6] = {'<', '<', '<', '<', '<', '<', '<', '<', '>', 'e'};
}

```

```

precedence[7] = {'>', '>', '>', '>', '>', 'e', 'e', '>', '>'};
precedence[8] = {'<', '<', '<', '<', '<', '<', '<', '<', '>'};

}

void printStack()
{
    for(int i = 0; i < top+1; i++)
        cout << st[i];
}

bool operator_pp(string s)
{
    cout << "\nSTACK \tInput
    ↪ \tAction\n===== \n";
    while(ptr < s.size())
    {
        shift(s[ptr]);
        printStack();
        cout << "\t" << s.substr(ptr);
        cout << "\t" << "Shift" << endl;

        int tp = getOperatorIndex(st[top]);
        int curr = getOperatorIndex(s[ptr]);

        if(precedence[tp][curr] == '>')
        {
            while(reduce())
            {
                printStack();
                cout << "\t" << s.substr(ptr);
                cout << "\tReduced : E->" << prevHandle <<
                ↪ endl;
            }
        }

    }

    if(st.size() > 1 && st[0] == '$' && st[1] == 'E' && st[2]
    ↪ == '$')
        return true;

    return false;
}

```

```

}

int main()
{
    string s;
    cin >> s;

    s += '$';

    setup();

    if(operator_pp(s))
        cout << "\nSuccessfully Parsed\n";
    else
        cout << "\nError in Expression\n";

    return 0;
}

```

Input & Output

```

Amriths-Air:Operator-precedence-parser amrithm98$ ./a.out
i+(i*i)

STACK   Input   Action
=====
$i      +(i*i)$  Shift
$E      +(i*i)$  Reduced : E->i
$E+     (i*i)$  Shift
$E+(    i*i)$  Shift
$E+(i   *i)$  Shift
$E+(E   *i)$  Reduced : E->i
$E+(E*  i)$  Shift
$E+(E*i)$  Shift
$E+(E*i)$  Reduced : E->i
$E+(E   )$  Reduced : E->E+E
$E+(E)  $        Shift
$E+E    $        Reduced : E->)E(
$E      $        Reduced : E->E+E
$E$     $        Shift

```

3.6 Program 2

Aim : Write program to Simulate First and Follow of any given grammar.

```

#include<bits/stdc++.h>
using namespace std;

set<char> terminals, nonTerminals;
map<char, set<string>> productions;

```

```
map<char, set<char>> first;
map<char, set<char>> follow;

set<string> visitedProductions;
set<char> visitedFollow;

void showDetails()
{
    cout << "\nProductions : \n";
    for(auto it : productions)
    {
        for(auto it1 : it.second)
            cout << it.first << " : " << it1 << endl;
    }

    cout << "\nSet of Terminals : ";
    for(auto it : terminals)
        cout << it << " ";

    cout << "\nSet of Non Terminals : ";
    for(auto it : nonTerminals)
        cout << it << " ";

    cout << "\nFIRST : \n";
    for(auto it : first)
    {
        cout << it.first << " : ";
        for(auto it1 : it.second)
            cout << it1 << " ";

        cout << endl;
    }

    cout << "\nFollow : \n";
    for(auto it : follow)
    {
        cout << it.first << " : ";
        for(auto it1 : it.second)
        {
            if(it1 != '#')
                cout << it1 << " ";
        }
        cout << endl;
    }
}
```

```
    }

    cout << "\n\n";
}

void findFirst(char c)
{
    for(auto it : productions[c])
    {
        if(terminals.find(it[0]) == terminals.end())
        {
            findFirst(it[0]);
            first[c].insert(first[it[0]].begin(),
                ↪ first[it[0]].end());
        }
        else
        {
            first[c].insert(it[0]);
        }
    }
}

void findFollow(char c)
{
    cout << "\nFollow of : " << c << endl;
    //Iterate through the right hand side of all productions
    ↪ Using 2 loops
    for(auto sets : productions)
    {
        for(auto it2 : sets.second)
        {
            //it2 represents one string which appears on the
            ↪ RHS of a grammar
            //Check if this production has been visited

            if(visitedProductions.find(it2) ==
                ↪ visitedProductions.end())
            {
                visitedProductions.insert(it2);
                //visit the production and see where the
                ↪ symbol 'c' appears

                for(int i = 0; i < it2.size(); i++)
```



```

{
    //Found that symbol in the RHS
    if(it2[i] == c)
    {
        //If it is the last symbol , find the
        → follow of the LHS
        if(i == it2.size()-1)
        {
            printf("\nFollow of %c is FOLLOW
            → of %c",c,sets.first);
            if(sets.first != c)
            {
                findFollow(sets.first);

                → follow[c].insert(follow[sets.first].begin(),
                → follow[sets.first].end());
            }
        }
        else if(i < it2.size()-1 &&
        → terminals.find(it2[i+1]) !=
        → terminals.end())
        {
            //If it is followed by a symbol,
            → insert that symbol into follow
            printf("\nFollow of %c Contains
            → %c",c,it2[i+1]);
            if(it2[i+1] != '#')
                follow[c].insert(it2[i+1]);
        }
        else
        {
            //Follow of 'c' is the First of
            → next non terminal it+1
            printf("\nFollow of %c is FIRST of
            → %c",c,it2[i+1]);

            → follow[c].insert(first[it2[i+1]].begin(),
            → first[it2[i+1]].end());

            //If First of i+1 Contains
            → Epsilon, find First of the next
            → symbol

```

```

//Do this iteratively because
↪ F->ABCDE, A contains #, take B,
↪ B contains # take C and so on
for(int k = i+1; k < it2.size();
    ↪ k++)
{
    if(first[it2[k]].find('#') !=
        ↪ first[it2[k]].end())
    {
        if(k < it2.size() - 1)
        {
            ↪ follow[c].insert(first[it2[k+1]]
            ↪ first[it2[k+1]].end());
        }
        else //If i+2th symbol
            ↪ didn't exist, find
            ↪ follow of LHS
        {
            ↪ findFollow(sets.first);

            ↪ follow[c].insert(follow[sets.f
            ↪ follow[sets.first].end());
        }
    }
}

}

}

}

visitedProductions.erase(it2);
}

}

}

}

int main()
{
    ifstream infile("productions.txt");

    string temp;

```

```
follow['E'].insert('$');

while(getline(infile,temp))
{
    stringstream ss(temp);
    string prods;

    vector<string> lhs_rhs;
    while(getline(ss,prods,'='))
    {
        lhs_rhs.push_back(prods);
    }

    productions[lhs_rhs[0][0]].insert(lhs_rhs[1]);

    for(int i = 0 ; i < temp.size(); i++)
    {
        if(isalpha(temp[i]) && temp[i] >= 65 && temp[i] <=
            ↪ 91)
            nonTerminals.insert(temp[i]);
        else
            terminals.insert(temp[i]);
    }
}

for(auto it : productions)
{
    findFirst(it.first);
}

for(auto it : nonTerminals)
{
    findFollow(it);
}

for(auto it : follow)
{
    if(it.second.find('#') != it.second.end())
        it.second.erase('#');
}

showDetails();
```

```

    return 0;
}

```

Input & Output

```

Productions :
E : TR
F : (E)
F : i
R : #
R : +TR
T : FY
Y : #
Y : *FY

Set of Terminals : # ( ) * + = i
Set of Non Terminals : E F R T Y
FIRST :
E : ( i
F : ( i
R : # +
T : ( i
Y : # *

Follow :
E : $ )
F : $ ) * +
R : $ )
T : $ ) +
Y : $ ) +

```

3.7 Program 3

Aim : : Construct a recursive descent parser for an expression.

```

#include<iostream>
#include<map>
#include<set>
#include<string>
#include<vector>
#include<fstream>
using namespace std;

map<char, vector<string>> productions;
set<char> terminals, non_terminals, all_symbols;

bool isTerminal(char c)
{
    if(terminals.find(c) == terminals.end())
        return false;

    return true;
}

```

```
bool isNonTerminal(char c)
{
    if(non_terminals.find(c) == non_terminals.end())
        return false;

    return true;
}

bool parse_input(int index , string &input, string production)
{
    // cout << input[index] << " " << production[0] << endl;

    if(production.size() > 0 && production[0] == '#')
        return parse_input(index, input,
            ↪ production.substr(1));

    if(production.size() == 0)
    {
        if(input[index] == '$')
            return true;

        return false;
    }

    if( isTerminal( production[0]) )
    {
        if(input[index] == production[0])
            return parse_input(index + 1, input,
                ↪ production.substr(1));
        else
            return false;
    }

    if( isNonTerminal(production[0]) )
    {
        char c = production[0];
        for(auto it : productions[c])
        {
            if(parse_input(index, input, it +
                ↪ production.substr(1)))
```

```
        return true;
    }
}

return false;
}

int main()
{
    fstream infile("grammar.txt");

    string start;
    getline(infile, start);

    string temp;

    while(getline(infile, temp))
    {
        productions[temp[0]].push_back(temp.substr(2));

        non_terminals.insert(temp[0]);

        all_symbols.insert(temp[0]);

        all_symbols.insert(temp.begin()+2, temp.end());
    }

    for(auto it : all_symbols)
    {
        if(non_terminals.find(it) == non_terminals.end())
            terminals.insert(it);
    }

    cout << "\nProductions \n===== \n";

    for(auto it : productions)
    {
        for(auto it1 : it.second)
            cout << it.first << " -> " << it1 << endl;
    }

    cout << "\nTerminals : ";
```

```

    for(auto it : terminals)
        cout << it << " ";

    cout << "\nNon Terminals : ";
    for(auto it : non_terminals)
        cout << it << " ";

    cout << "\nStart Symbol : " << start;

    string expression;

    cout << "\nEnter Expression : ";
    cin >> expression;

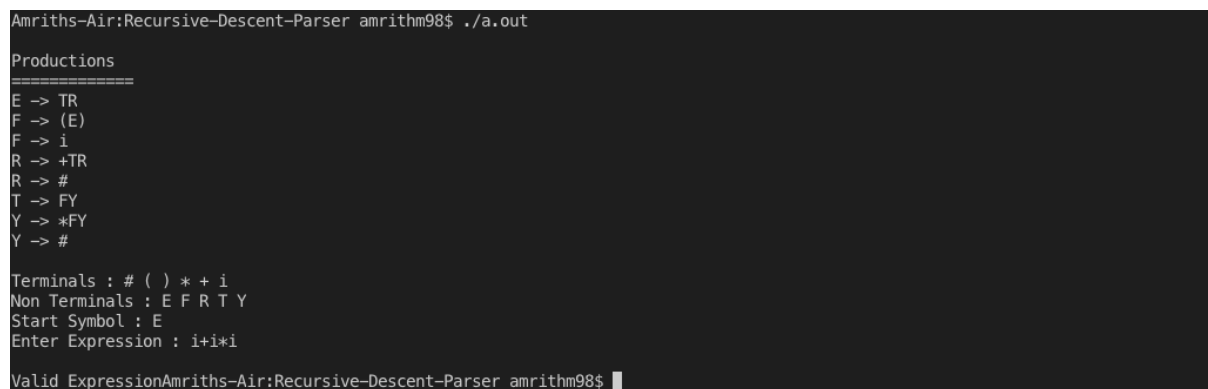
    expression.append("$");

    if(parse_input(0, expression, start))
        cout << "\nValid Expression\n";
    else
        cout << "\nInvalid Expression\n";

    return 0;
}

```

Input & Output



```

Amriths-Air:Recursive-Descent-Parser amrithm98$ ./a.out
Productions
=====
E -> TR
F -> (E)
F -> i
R -> +TR
R -> #
T -> FY
Y -> *FY
Y -> #

Terminals : # ( ) * + i
Non Terminals : E F R T Y
Start Symbol : E
Enter Expression : i+i*i

Valid ExpressionAmriths-Air:Recursive-Descent-Parser amrithm98$

```

3.8 Program 4

Aim : : Construct a Shift Reduce Parser for a given language.

```
#include <bits/stdc++.h>
using namespace std;

vector<char> st(100);
vector<string> handles;
vector<string> precedence(9);
string prevHandle;
int ptr;
int top;

int getOperatorIndex(char value)
{
    switch(value)
    {
        case '+':
            return 0;
        case '-':
            return 1;
        case '*':
            return 2;
        case '/':
            return 3;
        case '^':
            return 4;
        case 'i':
            return 5;
        case '(':
            return 6;
        case ')':
            return 7;
        case '$':
            return 8;
        default:
            return -1;
    }
}

void shift(char c)
{
    st[++top] = c;
    ptr++;
}
```



```
bool reduce()
{
    bool found;

    for(int i = 0; i < handles.size(); i++)
    {
        int size = handles[i].size();

        if(st[top] == handles[i][0] && top + 1 >= size )
        {
            found = true;
            int j;
            for(j = 0; j < size; j++)
            {
                if(st[top - j] != handles[i][j])
                {
                    found = false;
                    break;
                }
            }

            if(found)
            {
                st[top - j + 1] = 'E';
                top = top - j + 1;
                prevHandle = handles[i];
                return true;
            }
        }
    }

    return false;
}

void setup()
{
    ptr = 0;
    st[0] = '$';
    handles = {"i", "E+E", "E*E", "E-E", "E^E", ")E("};
    top = 0;
    //Instead of USING this table to shift , Keep shifting all
    ↪ the time
    //Precedence Table
```

```

precedence[0] = {'>', '>', '<', '<', '<', '<', '<', '<', '>', '>'};
precedence[1] = {'>', '>', '<', '<', '<', '<', '<', '<', '>', '>'};
precedence[2] = {'>', '>', '>', '>', '<', '<', '<', '<', '>', '>'};
precedence[3] = {'>', '>', '>', '>', '<', '<', '<', '<', '>', '>'};
precedence[4] = {'>', '>', '>', '>', '<', '<', '<', '<', '>', '>'};
precedence[5] = {'>', '>', '>', '>', '>', 'e', 'e', '>', '>'};
precedence[6] = {'<', '<', '<', '<', '<', '<', '<', '<', '>', 'e'};
precedence[7] = {'>', '>', '>', '>', '>', 'e', 'e', '>', '>'};
precedence[8] = {'<', '<', '<', '<', '<', '<', '<', '<', '<', '>'};

}

void printStack()
{
    for(int i = 0; i < top+1; i++)
        cout << st[i];
}

bool operator_pp(string s)
{
    cout << "\nSTACK \tInput
    ↪ \tAction\n===== \n";
    while(ptr < s.size())
    {
        shift(s[ptr]);
        printStack();
        cout << "\t" << s.substr(ptr);
        cout << "\t" << "Shift" << endl;

        int tp = getOperatorIndex(st[top]);
        int curr = getOperatorIndex(s[ptr]);

        if(precedence[tp][curr] == '>')
        {
            while(reduce())
            {
                printStack();
                cout << "\t" << s.substr(ptr);
                cout << "\tReduced : E->" << prevHandle <<
                ↪ endl;
            }
        }
    }
}

```

```

    }

    if(st.size() > 1 && st[0] == '$' && st[1] == 'E' && st[2]
    → == '$')
        return true;

    return false;
}

int main()
{
    string s;
    cin >> s;

    s += '$';

    setup();

    if(operator_pp(s))
        cout << "\nSuccessfully Parsed\n";
    else
        cout << "\nError in Expression\n";

    return 0;
}

```

Input & Output

```

Amriths-Air:Operator-precedence-parser amrithm98$ ./a.out
i+(i*i)

STACK  Input  Action
=====
$i    +(i*i)$  Shift
$E    +(i*i)$  Reduced : E->i
$E+   (i*i)$  Shift
$E+(  i*i)$  Shift
$E+(i  *i)$  Shift
$E+(E  *i)$  Reduced : E->i
$E+(E* i)$  Shift
$E+(E*i )$  Shift
$E+(E*E )$  Reduced : E->i
$E+(E  )$  Reduced : E->E+E
$E+(E)  $  Shift
$E+E   $  Reduced : E->)E(
$E     $  Reduced : E->E+E
$E$    $  Shift

```

4 Cycle -4 (Compiler & Interpreter)

4.1 Using Lex and YACC to build a Compiler

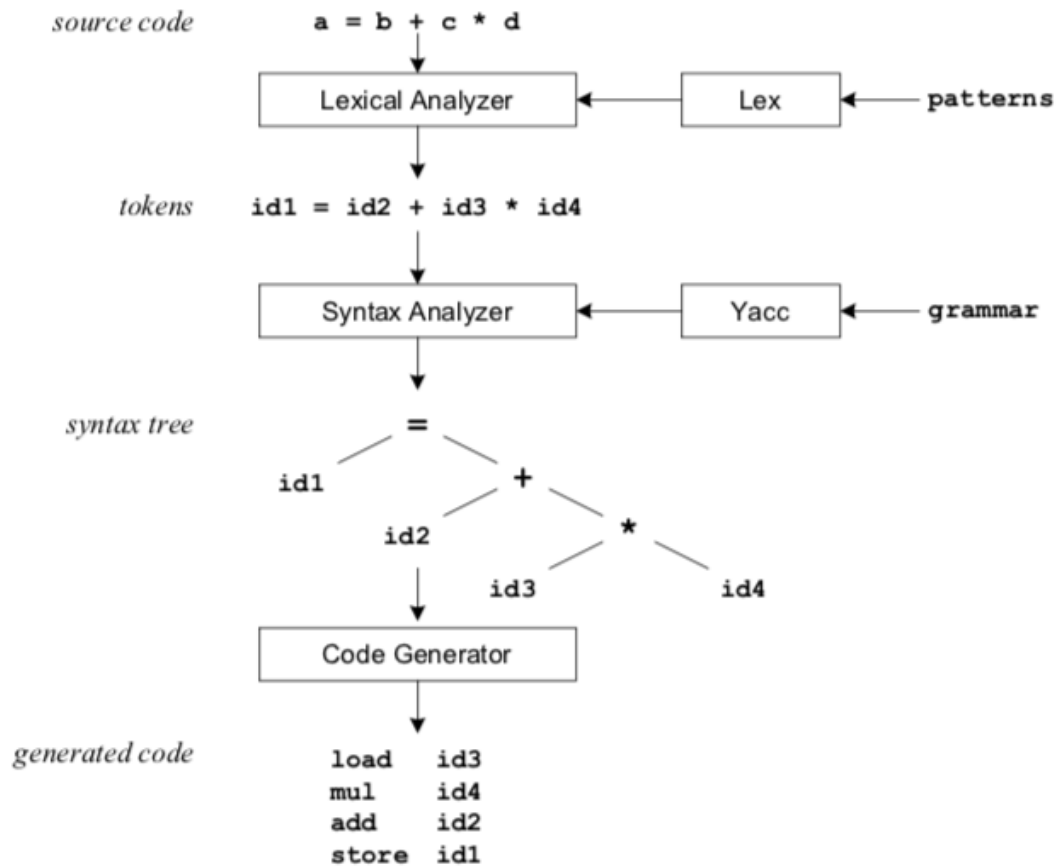


Figure 1: Compilation Sequence

The patterns in the above diagram is a file you create with a text editor. Lex will read your patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing. When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index. The grammar in the above diagram is a text file you create with a text editor. Yacc will read your grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree.

The syntax tree imposes a hierarchical structure the tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly language.

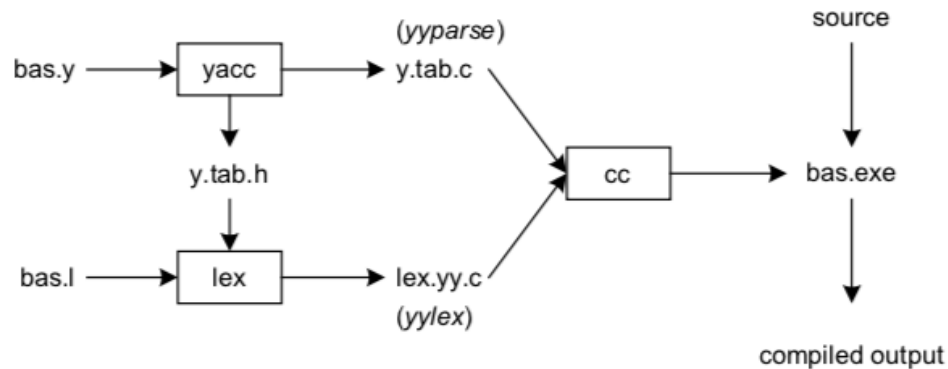


Figure 2: Building a Compiler with Lex/Yacc

Figure 2 illustrates the file naming conventions used by lex and yacc. We'll assume our goal is to write a BASIC compiler. First, we need to specify all pattern matching rules for lex (bas.l) and grammar rules for yacc (bas.y). Commands to create our compiler, bas.exe, are listed below:

```

yacc -d bas.y          # create y.tab.h, y.tab.c
lex bas.l              # create lex.yy.c
cc lex.yy.c y.tab.c -obas.exe # compile/link
  
```

4.2 Program 1,2,3

In this lab we are going to construct a compiler—interpreter for a simple imperative programming language. The CFG is given below.

Program → PROG declarations BEGIN command_sequence END

declarations → e | INTEGER id_seq IDENTIFIER.

id_seq → e | id_seq IDENTIFIER,

command_sequence → e | command_sequence command ;

command → e

| IDENTIFIER : =expression

| IF exp THEN command_sequence ELSE command_sequence ENDIF

| WHILE exp DO command_sequence ENDWHILE

| READ IDENTIFIER

| WRITE expression

expression → NUMBER | IDENTIFIER | (expression)

| expression + expression | expression * expression

| expression - expression | expression / expression

| expression = expression

| expression < expression

| expression > expression

where the non-terminal symbols are given in all lowercase and the terminal symbols are given in all caps or as literal symbols. The start symbol is program. There are two context sensitive requirements for the language, variables must be declared before they are referenced and a variable may be declared only once.

```

prog
    integer a,b.
begin
    read n;
    if a < 10 then b := 1; else; endif;
    while a < 10 do b := 5*a; a:= a+1; endwhile;
    write a;
    write b;
end

```

4.3 Symbol Table

Structure of the symbol table is

```

struct sym_rec {
    char *name;           //name of the symbol
    struct sym_rec next;  //link field
    int data_offset;      //used during code generation
    ↪ phase
} *sym_record;           //points to the first record

struct sym_rec * put_symbol(char * name);
struct sym_rec * get_symbol(char * name);
void install(char *name);
void context_check(char *name);

```

- **put_symbol()** : puts an identifier into the table.
- **get_symbol()** : returns a pointer to the symbol table entry or a NULL pointer if not found.
- **install()** : installs a symbol into the symbol table if it is not in the symbol table using the above two functions. Reports appropriate error messages.
- **context_check()**: checks the context sensitive requirement of our language and if violated appropriate error messages

4.4 Code Genenration

The code is generated from the implicitly created parse tree. Here we are generating code for a virtual machine called a stack machine. The virtual machine consists of three segments. A **data segment**, a **code segment** and an **expression stack**. The data segment contains the values associated with the variables. Each variable is assigned to a location in the data segment which holds the associated value. The code segment consists of a sequence of operations, i.e. the stack machine code. Program constants are incorporated in the code segment since their values do not change. The expression stack is a stack which is used to hold intermediate values in the evaluation of expressions.

Instruction format for the stack machine:

opcode	operand
--------	---------

Instruction set (For generating Machine Code) :

instruction	operand	meaning
res	n	Reserve n locations in the bottom of the stack
read	n	stack(n) := input
write	0	output := stack(top- -)
goto	n	Execute from the nth code onwards in the code segment
lt	0	If stack(top-1) < stack(top) stack(top) := 1 else stack(top) := 0
gt	0	If stack(top-1) > stack(top) stack(top) := 1 else stack(top) := 0
eq	0	If stack(top-1) == stack(top) stack(top) := 1 else stack(top) := 0
jmp_false	n	If stack(top) = 0 execute the nth code in the code segment, top- -
load_var	n	load variable at base offset n of the stack into top of the stack, top- -
load_int	n	stack(++top) := n
store	n	store stack(top) at variable location n from base of the stack, top- -
add	0	stack(top-1) := stack(top) + stack(top-1), top- -
mul	0	stack(top-1) := stack(top) * stack(top-1), top- -
div	0	stack(top-1) := stack(top) / stack(top-1), top- -
sub	0	stack(top-1) := stack(top) - stack(top-1), top- -
halt	0	stop execution

Translation Schemes(For use in Interpreter)

declaration statements :

integer x , y , z : res 3

statements :

x := expression	:	code for expression store x
read x	:	read x
write expression	:	code for expression write
WHILE expression DO command_sequence ENDWHILE	:	L1 : code for expression jmp_false L2 code for command_sequence goto L1 L2:
IF expression THEN comand_sequence1 ELSE command_sequence2 ENDIF	:	code for expression jmp_false L1 code for command_sequence1 goto L2 L1 : code for command_sequence2 L2 :

expressions :

constant	:	load_int constant
variable	:	load_var variable
expression1 op expression2	:	code for expression1 code for expression2 code for op

Aim :

- Generation of lexical analyzer and parser for our language.

- Creating a symbol table for our language, (linked list implementation)
- Code generation.

Lex Code

```

/** Definition section */

%{
/* C code to be copied verbatim */
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"
#include <iostream>
using namespace std;
int input_line_no = 1;
%}

/* This tells flex to read only one input file */
%option noyywrap
%%
[\\n]                {input_line_no++;}
"prog"              return PROG;
"integer"           return INT;
"begin"             return BEG;
"read"             return READ;
"if"               return IF;
"then"             return THEN;
"else"             return ELSE;
"endif"           return EIF;
"while"           return WHILE;
"do"             return DO;
"endwhile"       return EWHILE;
"write"         return WRITE;
"end"           return END;
[a-zA-Z_][_a-zA-Z0-9]* {
    yylval.name = strdup(yytext);
    return IDENTI;
}
[0-9]+            {
    yylval.num = atoi(yytext);
    return NUM;
}
"<"             return LS;

```

```

">"          return GT;
"."          return DOT;
","          return COMMA;
":="         return EQUAL;
"+"          return PLUS;
"-"          return MINUS;
"*"          return MULT;
"/"          return DIV;
";"          return SEMICOLON;

```

```
%%
```

YACC Code

```

%{
    #include<stdio.h>
    #include<iostream>
    using namespace std;

    int yylex();
    void yyerror(const char *s);

    void printError(int code);
    void write_machine_code();
    extern int input_line_no;

    int data_offset = 0;

    struct sym_rec
    {
        char *name; //name of the symbol
        int data; //Data
        struct sym_rec *next; //link field
        int data_offset; //will be used during code generation
        ↪ phase.
    };

    struct sym_rec *sym_record;

    struct sym_rec * put_symbol(char *name, int data);
    struct sym_rec * get_symbol(char *name);
    void install(char *name);
    void displaySymTab();
    void context_check(char *name);

```

```
char machine_code[1000];
int pos = 0;
int output_line_no = 1;

struct stack_node{
    int pos;
    struct stack_node* next;
};

stack_node *stack_top = 0;
void push(int pos);
int pop();
void replace(char str[], int pos , int n);
void write_machine_code_to_file(const char*);
%}

%union
{
    char *name;
    int num;
    int offset;
}

%token PROG
%token INT
%token BEG
%token READ
%token IF
%token THEN
%token ELSE
%token EIF
%token WHILE
%token DO
%token EWHILE
%token WRITE
%token END
%token LS
%token DOT
%token COMMA
%token EQUAL
%token PLUS
%token MINUS
```

```

%token MULT
%token DIV
%token SEMICOLON
%token GT

%token NUM IDENTI
%type<num> expression NUM
%type<name> IDENTI

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%left '<' '>'

%start Pro
%%

```

```

Pro:    PROG declarations
        {
            pos += sprintf( machine_code + pos , "res\t\t%d\n"
                ↪ , data_offset/4);
            output_line_no++;
        }
        BEG command_sequence END
        {
            pos += sprintf( machine_code + pos ,
                ↪ "halt\t\t0\n");
            output_line_no++;
        }

```

```

declarations: INT id_seq IDENTI DOT
        {
            install($3);
        }
        |
        ;

```

```

id_seq: id_seq IDENTI COMMA
        {
            install($2);
        }

```

```

}
|
;

```

```

command_sequence:command_sequence command SEMICOLON

```

```

|
;

```

```

command : IDENTIFIER EQUAL expression

```

```

{
    context_check($1);
    int offset = get_symbol($1)->data_offset/4;
    pos += sprintf(machine_code + pos ,
        ↪ "store\t\t%d\n" , offset);
    put_symbol($1, $3);
    output_line_no++;
}

| IF expression THEN
{
    //Push 000, replace it with next line of goto
    push(pos + 11);
    pos += sprintf(machine_code + pos ,
        ↪ "jmp_false\t\t000\n");
    output_line_no++;
}

command_sequence
{
    //Line after goto appear here
    replace(machine_code, pop(), output_line_no + 1);
    //Goto 000, replace it once you get 000
    push(pos + 6);

    pos += sprintf(machine_code + pos ,
        ↪ "goto\t\t000\n");
    output_line_no++;
}

ELSE command_sequence EIF
{
    replace(machine_code, pop(), output_line_no);
}

```

```

| WHILE
{
    push(output_line_no);
}
expression DO
{
    push(pos + 11);
    pos += sprintf(machine_code+pos ,
        ↪ "jmp_false\t\t000\n");
    output_line_no++;
}

command_sequence
{
    int replace_pos = pop();
    pos += sprintf(machine_code+pos ,
        ↪ "goto\t\t%03d\n", pop());
    output_line_no++;
    replace(machine_code, replace_pos,
        ↪ output_line_no);
}
EWHILE

| READ IDENTIFIER
{
    context_check($2);
    int offset = get_symbol($2)->data_offset/4;
    pos += sprintf(machine_code + pos , "read\t\t%d\n"
        ↪ ,offset);
    output_line_no++;
}

| WRITE expression
{
    pos += sprintf(machine_code+pos , "write\t\t0\n");
    output_line_no++;
}
|
;

expression : NUM
{

```

```

    pos += sprintf(machine_code+pos ,
        ↪ "load_int\t\t%d\n" , $1);
    $$ = $1;
    output_line_no++;
}

| IDENTIFIER
{
    int offset = get_symbol($1)->data_offset/4;
    pos += sprintf(machine_code+pos ,
        ↪ "load_var\t\t%d\n" , offset);
    $$ = (get_symbol($1) != NULL) ?
        ↪ get_symbol($1)->data:0;
    output_line_no++;
}

| '(' expression ')'
{
    $$ = $2;
}

| expression PLUS expression
{
    pos += sprintf(machine_code+pos , "add\t\t0\n");
    $$ = $1+$3;
    output_line_no++;
}

| expression MULT expression
{
    pos += sprintf(machine_code+pos , "mul\t\t0\n");
    $$ = $1*$3;
    output_line_no++;
}

| expression MINUS expression
{
    pos += sprintf(machine_code+pos , "sub\t\t0\n");
    $$ = $1-$3;
    output_line_no++;
}

| expression DIV expression

```



```

    {
        pos += sprintf(machine_code+pos , "div\t\t0\n");
        $$ = $1/$3;
        output_line_no++;
    }

| expression LS expression
{
    pos += sprintf(machine_code+pos , "lt\t\t0\n");
    $$ = $1<$3;
    output_line_no++;
}

| expression GT expression
{
    pos += sprintf(machine_code+pos , "gt\t\t0\n");
    $$ = $1>$3;
    output_line_no++;
}
;

%%

// struct sym{
//     char name;
//     int val;
// };

void yyerror(const char*)
{
    // fprintf(stderr, "error: %s\n", str);
}

void printError(int err)
{
    switch(err)
    {
        case 1 :
            printf("\nError: %d -> Compilation Error Found"
                , input_line_no);
            break;
    }
}

```

```
        case 2:
            printf("\nERROR: %d -> Redefinition Of Variable\n",
                input_line_no);
            break;
        case 3:
            printf("\nERROR: %d -> Undefined Variable\n",
                input_line_no);
            break;
    }
    exit(0);
}

int yywrap()
{
    return 1;
}

void context_check(char *name)
{
    if(get_symbol(name) == NULL)
        printError(3);
}

struct sym_rec * get_symbol(char *name)
{
    struct sym_rec *temp = sym_record;
    while(temp != NULL)
    {
        if(strcmp(temp->name, name) == 0)
            return temp;

        temp = temp->next;
    }

    return NULL;
}

void install(char *name)
{
    struct sym_rec *temp = get_symbol(name);

    if(temp == NULL)
```

```
{
    temp = (struct sym_rec*)malloc(sizeof(struct
        ↪ sym_rec));
    temp->name = name;
    temp->data = 0;
    temp->next = sym_record;
    temp->data_offset = data_offset;
    data_offset = data_offset + 4;
    sym_record = temp;
}
else
    printError(2);
}

struct sym_rec *put_symbol(char *name, int data)
{
    struct sym_rec *temp = get_symbol(name);

    if(temp == NULL)
    {
        cout << "\nERROR: Undefined Variable " << name <<
            ↪ endl;
        return NULL;
    }

    temp->data = data;

    return temp;
}

void displaySymTab()
{
    cout << "\nSymbol Table : ";

    struct sym_rec *temp = sym_record;
    while(temp != NULL)
    {
        if(strcmp(temp->name, "end") != 0)
            cout << "\nName : " << temp->name << " Value : "
                << temp->data << " Offset: "
                << temp->data_offset << "\n";
        temp = temp->next;
    }
}
```

```
}

void write_machine_code()
{
    int line_no = 0;
    printf("\nStack Machine Code : \n\n");
    for(int i = 0 ; machine_code[i] != '\0' ; i++){
        if(i == 0 || machine_code[i-1] == '\n'){
            line_no++;
            printf("%03d : ", line_no);
        }
        printf("%c", machine_code[i]);
    }
}

void write_machine_code_to_file(const char* filename){
    FILE *output = fopen(filename , "w");
    fprintf(output , "%s", machine_code);
    fclose(output);
}

void push(int pos){
    struct stack_node *node = (struct stack_node*)malloc
        (sizeof(struct stack_node));
    node->pos = pos;
    node->next = stack_top;
    stack_top = node;
}

int pop(){
    if(stack_top == 0)
        return -1;
    int pos = stack_top->pos;
    struct stack_node* node = stack_top;
    stack_top = stack_top->next;
    free(node);
    return pos;
}

void replace(char str[] , int pos , int n){
    str[pos] = n/100 + '0';
    str[pos+1] = (n%100) / 10 + '0';
    str[pos+2] = (n%10) + '0';
}
```

```

}

int main()
{
    sym_record = (struct sym_rec *)malloc
        (sizeof(struct sym_rec));
    sym_record->name = "end";
    sym_record->data = -1;
    sym_record->next = NULL;

    yyparse();
    printf("\nThe program was successfully
    parsed and accepted\n");
    displaySymTab();
    write_machine_code();
    write_machine_code_to_file("machine_code");
    return 0;
}

```

Input & Output

```
Amriths-Air:Parser amrithm98$ ./mc < inp.i
```

```
The program was successfully parsed and accepted
```

```
Symbol Table :
Name : b Value : 0 Offset: 4
Name : a Value : 1 Offset: 0
```

```
Stack Machine Code :
```

```

001 : res      2
002 : read     0
003 : load_var 0
004 : load_int 10
005 : lt       0
006 : jmp_false 010
007 : load_int 1
008 : store    1
009 : goto     010
010 : load_var 0
011 : load_int 10
012 : lt       0
013 : jmp_false 023
014 : load_int 5
015 : load_var 0
016 : mul      0
017 : store    1
018 : load_var 0
019 : load_int 1
020 : add      0
021 : store    0
022 : goto     010
023 : load_var 0
024 : write    0
025 : load_var 1
026 : write    0
027 : halt     0

```

4.5 Program 4

Aim : : Design an interpreter for the stack machine.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define STACK_SIZE 100
#define CODE_SIZE 100

#define RES      1
#define READ     2
#define WRITE    3
#define GOTO     4
#define LT       5
#define GT       6
#define EQ       7

#define JMP_FALSE 8
#define LOAD_INT  9
#define LOAD_VAR  10

#define STORE     11
#define ADD       12
#define MUL       13
#define DIV       14
#define SUB       15
#define HALT      16

struct instruction{

    char opcode[10];
    int operand;
};

struct instruction code_seg[CODE_SIZE];
int* data_seg = 0;
int stack[STACK_SIZE];
int top = -1;

int pop();
```

```
void push(int);
int convert_opcode(char[]);

int main(int argc , char** argv)
{
    char opcode[10];
    int operand;
    int temp;
    int line = 1;
    int code_size = 0;
    int left , right;
    FILE *input;

    if(strcmp(argv[1] , "stdin") == 0)
        input = stdin;
    else
        input = fopen(argv[1] , "r");

    while(1) {

        if(line > code_size) {

            code_size++;
            fscanf(input, "%s %d" , code_seg[code_size].opcode
                ↪ , &code_seg[code_size].operand);
            continue;
        }

        strcpy(opcode , code_seg[line].opcode);
        operand = code_seg[line].operand;

        printf("%d %s %d\n" , line , opcode , operand);

        switch(convert_opcode(opcode)) {

            case RES : data_seg = (int*) malloc(operand *
                ↪ sizeof(int));
                        break;

            case READ : scanf("%d" , &temp);
                        data_seg[operand] = temp;
                        break;
        }
    }
}
```

```
case WRITE : printf("%d\n" , pop());
             break;

case GOTO : line = operand - 1; // line++
           ↪ after switch()
             break;

case LT : right = pop();
         left = pop();

         if(left < right)
             push(1);
         else
             push(0);

         break;

case GT : right = pop();
         left = pop();

         if(left > right)
             push(1);
         else
             push(0);

         break;

case EQ : right = pop();
         left = pop();

         if(left == right)
             push(1);
         else
             push(0);

         break;

case JMP_FALSE : if( pop() == 0 )
                  line = operand-1; //line++
                  ↪ after switch()
                  break;
```



```
        case LOAD_VAR    :    push(data_seg[operand]);
                               break;

        case LOAD_INT    :    push(operand);
                               break;

        case STORE       :    data_seg[operand] = pop();
                               break;

        case ADD          :    right = pop();
                               left  = pop();
                               push( left + right );
                               break;

        case MUL          :    right = pop();
                               left  = pop();
                               push( left * right );
                               break;

        case DIV          :    right = pop();
                               left  = pop();
                               push( left / right );
                               break;

        case SUB          :    right = pop();
                               left  = pop();
                               push( left - right );
                               break;

        case HALT        :    exit(0);

        default           :    printf("Error : Unknown
↪ command...\n");
    }

    line++;
}

return 0;
}

int pop() {
```

```
    if(top == -1){

        printf("Error : stack underflow...\n");
        exit(0);
    }

    top--;

    return stack[top+1];
}

void push(int n){

    if(n == STACK_SIZE-1){

        printf("Error : stack overflow...\n");
        exit(0);
    }

    top++;

    stack[top] = n;

}

int convert_opcode(char opcode[]){

    if(strcmp(opcode , "res") == 0)
        return RES;

    if(strcmp(opcode , "read") == 0)
        return READ;

    if(strcmp(opcode , "write") == 0)
        return WRITE;

    if(strcmp(opcode , "goto") == 0)
        return GOTO;

    if(strcmp(opcode , "lt") == 0)
        return LT;
```

```
    if(strcmp(opcode , "eq") == 0)
        return EQ;

    if(strcmp(opcode , "gt") == 0)
        return GT;

    if(strcmp(opcode , "jmp_false") == 0)
        return JMP_FALSE;

    if(strcmp(opcode , "load_var") == 0)
        return LOAD_VAR;

    if(strcmp(opcode , "load_int") == 0)
        return LOAD_INT;

    if(strcmp(opcode , "store") == 0)
        return STORE;

    if(strcmp(opcode , "add") == 0)
        return ADD;

    if(strcmp(opcode , "mul") == 0)
        return MUL;

    if(strcmp(opcode , "div") == 0)
        return DIV;

    if(strcmp(opcode , "sub") == 0)
        return SUB;

    if(strcmp(opcode , "halt") == 0)
        return HALT;

    return -1;
}
```

Input & Output

Stack Machine Code :

```
001 : res      2
002 : read     0
003 : load_var 0
004 : load_int 10
005 : lt       0
006 : jmp_false 010
007 : load_int 1
008 : store    1
009 : goto     010
010 : load_var 0
011 : load_int 10
012 : lt       0
013 : jmp_false 023
014 : load_int 5
015 : load_var 0
016 : mul      0
017 : store    1
018 : load_var 0
019 : load_int 1
020 : add      0
021 : store    0
022 : goto     010
023 : load_var 0
024 : write    0
025 : load_var 1
026 : write    0
027 : halt     0
5
10
45
```

Script that takes a program as input and produces the executed results :

```
#!/bin/bash
rm lex.yy.c
rm interp
rm mc
rm y.tab.c
rm y.tab.h

lex comp.l
yacc -d comp.y
g++ lex.yy.c y.tab.c -o mc
./mc < inp.i

gcc interpreter.c -o interp
./interp machine_code|
```