Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 1

Дисциплина Конструирование компиляторов

Тема Распознавание цепочек регулярного языка

Вариант №5

Студент Золотухин А. В.

Группа ИУ7-21М

Преподаватель Ступников А.А.

Москва.
2025 г.

Задание

Напишите программу, которая в качестве входа принимает произвольное регулярное выражение, и выполняет следующие преобразования:

1) Преобразует регулярное выражение непосредственно в ДКА.

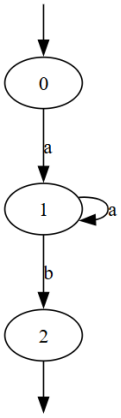
2) По ДКА строит эквивалентный ему КА, имеющий наименьшее возможное количество состояний.

Указание. Воспользоваться минимизацией ДКА, алгоритм за $O(n^2)$ с построением пар различимых состояний.

3) Моделирует минимальный КА для входной цепочки из терминалов исходной грамматики.

Результаты и выводы

Минимальный ДКА



Входные данные		**Результат**
Рег.выражение	**Строка**	
a+b	aaaab	|-aaaab|-aaab|-aab|-ab|-b|-да
	bbaaa	|-bbaaa|-нет
	abab	|-abab|-bab|-ab|-нет
	ab	|-ab|-b|-да
	abc	|-abc|-bc|-c|-нет

Контрольные вопросы

1. Какие из следующих множеств регулярны? Для тех, которые регулярны, напишите регулярные выражения.
    a. Множество цепочек с равным числом нулей и единиц.

Не является регулярным множеством (возможно контекстно-зависимая грамматика?)

b. Множество цепочек из {0, 1}* с четным числом нулей и нечетным числом единиц.

   Нет

c. Множество цепочек из {0, 1}*, длины которых делятся на 3.

   ((0|1)(0|1)(0|1))*

d. Множество цепочек из {0, 1}*, не содержащих подцепочки 101.

   0*(1|00+)*0*

2. Найдите праволинейные грамматики для тех множеств из вопроса 1, которые регулярны.

c	d
S → A A → 0B A → 1B A → ε B → 0C B → 1C C → 0A C → 1A	S → A A → 0A A → B B → 1B B → 0C C → B C → 0C B → D D → 0D D → ε

3. Найдите детерминированные и недетерминированные конечные автоматы для тех множеств из вопроса 1, которые регулярны

   c.

   НКА

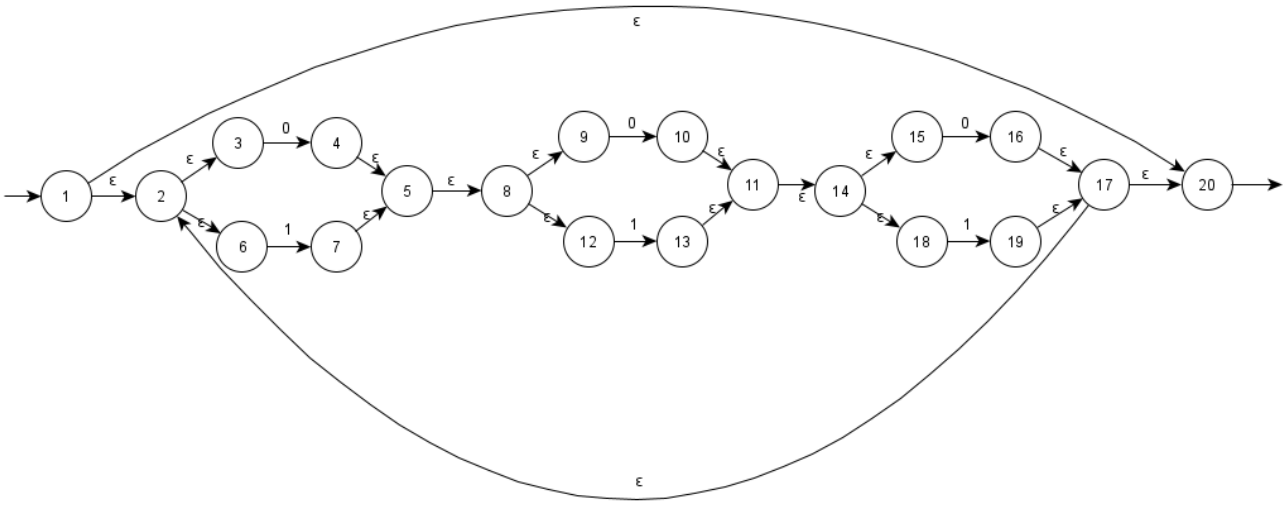


Рисунок 1 – НКА 3c

ДКА

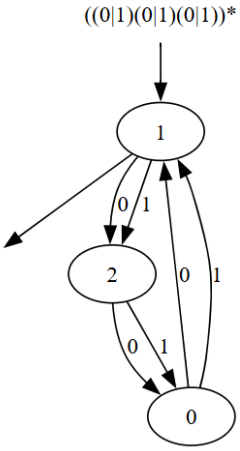


Рисунок 2 -- ДКА 3c

d.

НКА

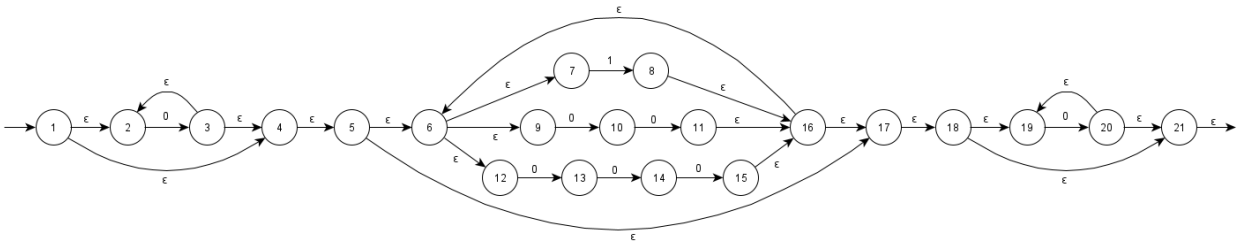


Рисунок 3 -- 3d

ДКА

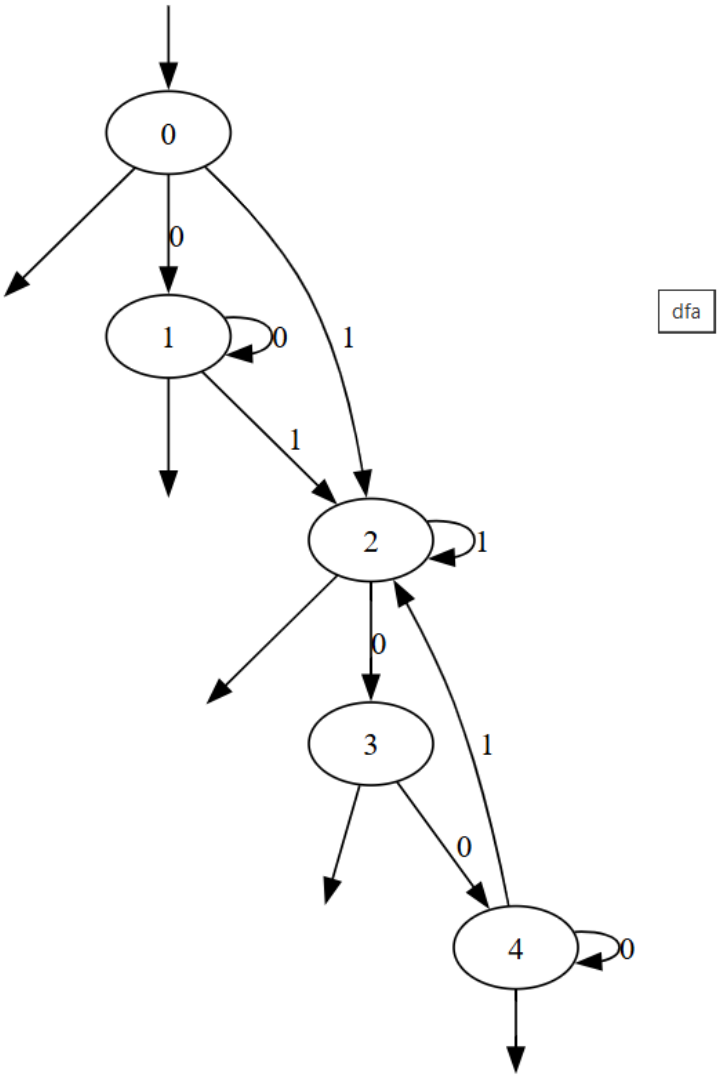


Рисунок 4 -- ДКА 3d

4. Найдите конечный автомат с минимальным числом состояний для языка, определяемого автоматом M = ({A, B, C, D, E}, {0, 1}, d, A, {E, F}), где функция задается таблицей

Состояние	Вход	
	0	1
A	B	C
B	E	F
C	A	A
D	F	E
E	D	F
F	D	E

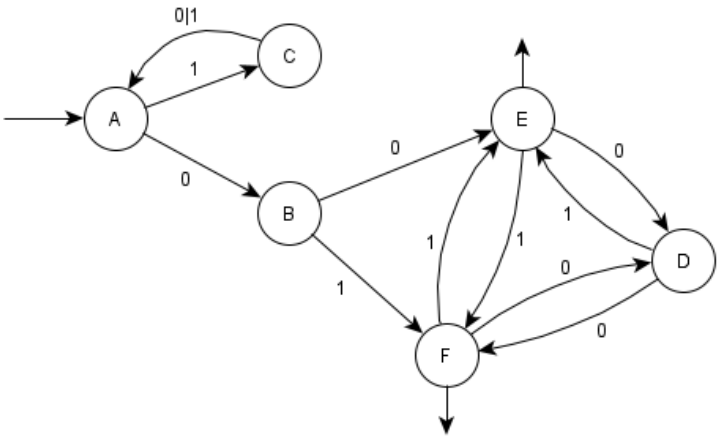


Рисунок 5 -- 4 задание

Использовался метод различимых состояний.

Таблица неэквивалентности:

	A	B	C	D	E	F
A		■	■	■	■	■
B	■		■		■	■
C	■	■		■	■	■
D	■		■		■	■
E	■	■	■	■		
F	■	■	■	■		

Вектор классов эквивалентности:

A	B	C	D	E	F
0	1	2	1	3	3

Стартовая вершина: A

Терминальная вершина: E

Минимальный КА:

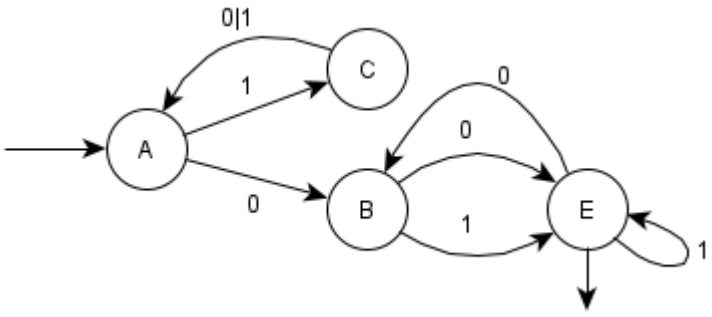


Рисунок 6 -- Минимальный КА

Текст программы

Листинг 1 – "Классы узлов дерева"

```
public class Node(string value, Node? leftNode = null, Node?
   rightNode = null)
{
    public int index;
    public string value = value;
    public Node? leftChild = leftNode;
    public Node? rightChild = rightNode;
    public bool nullable = false;
    public HashSet<int> firstpos = [], lastpos = [];
    public Dictionary<int, HashSet<int>> followpos = [];
    public DFA CreateDFA()
    {
        SetIndex(1);
        NullableFirstposLastpos();
        Followpos();
        Dictionary<int, HashSet<int>> treeFollowpos = [];
        treeFollowpos = GetTreeFollowpos(treeFollowpos);
        Dictionary<int, string> indexedStates = [];
        indexedStates = GetIndexedStates(indexedStates);
        return new DFA(this, indexedStates, treeFollowpos);
    }

    public int SetIndex(int index)
    {
        if (leftChild is null && rightChild is null)
        {
            this.index = index;
            index++;
        }
        else
        {
            if (leftChild is not null)
                index = leftChild.SetIndex(index);
            if (rightChild is not null)
                index = rightChild.SetIndex(index);
        }
        return index;
    }
```

```
public Dictionary<int, HashSet<int>>
   GetTreeFollowpos(Dictionary<int, HashSet<int>> res)
{
    foreach (var pair in followpos)
    {
        if (res.TryGetValue(pair.Key, out HashSet<int>?
           value))
        {
            res[pair.Key] = [.. value, .. pair.Value];
        }
        else
        {
            res[pair.Key] = [ .. pair.Value];
        }
    }
    if (leftChild is not null)
        res = leftChild.GetTreeFollowpos(res);
    if (rightChild is not null)
        res = rightChild.GetTreeFollowpos(res);
    return res;
}

private Dictionary<int, string>
   GetIndexedStates(Dictionary<int, string> res)
{
    if (leftChild is null && rightChild is null)
    {
        res[index] = value;
    }
    else
    {
        if (leftChild is not null)
            res = leftChild.GetIndexedStates(res);
        if (rightChild is not null)
            res = rightChild.GetIndexedStates(res);
    }
    return res;
}

public virtual void NullableFirstposLastpos()
```

```
    {
        if (string.IsNullOrEmpty(value))
        {
            nullable = true;
        }
        else
        {
            nullable = false;
            firstpos.Add(index);
            lastpos.Add(index);
        }
    }
    public virtual void Followpos()
    {
        leftChild?.Followpos();
        rightChild?.Followpos();
    }
}

public class NodeOperator(string value, Node? leftNode = null,
   Node? rightNode = null) : Node(value, leftNode, rightNode)
{

}

public class NodeStar(Node? leftNode = null, Node? rightNode =
   null) : Node(Consts.starSymbol, leftNode, rightNode)
{
    public override void NullableFirstposLastpos()
    {
        leftChild?.NullableFirstposLastpos();
        nullable = true;
        firstpos = [.. leftChild!.firstpos];
        lastpos = [.. leftChild!.lastpos];
    }

    public override void Followpos()
    {
        leftChild?.Followpos();
        foreach (var item in leftChild!.lastpos)
        {
```

```
            followpos[item] = [.. leftChild.firstpos];
        }
    }

}

public class NodeOr(Node ? leftNode = null, Node ? rightNode =
   null) : Node(Consts.orSymbol, leftNode, rightNode)
{
    public override void NullableFirstposLastpos()
    {
        leftChild?.NullableFirstposLastpos();
        rightChild?.NullableFirstposLastpos();
        nullable = leftChild!.nullable || rightChild!.nullable;
        firstpos = [.. leftChild!.firstpos, ..
           rightChild!.firstpos];
        lastpos = [.. leftChild!.lastpos, ..
           rightChild!.lastpos];
    }
}

public class NodeAnd(Node? leftNode = null, Node? rightNode =
   null) : Node(Consts.andSymbol, leftNode, rightNode)
{
    public override void NullableFirstposLastpos()
    {
        leftChild?.NullableFirstposLastpos();
        rightChild?.NullableFirstposLastpos();

        nullable = leftChild!.nullable && rightChild!.nullable;

        firstpos = [.. leftChild.firstpos];
        if (leftChild.nullable)
            firstpos = [.. firstpos, .. rightChild!.firstpos];

        lastpos = [.. rightChild!.lastpos];
        if (rightChild.nullable)
            lastpos = [.. lastpos, .. leftChild!.lastpos];
    }

    public override void Followpos()
```

```
    {
        leftChild?.Followpos();
        rightChild?.Followpos();
        foreach (var item in leftChild!.lastpos)
            followpos[item] = [.. rightChild!.firstpos];
    }
}
```

Листинг 2 – "Класс ДКА"

```
public class DFA
{
    public int startState = 0;
    public HashSet<int> finishStates = [];
    public List<DFAState> Dstates = [];
    public Dictionary<int, Dictionary<string, int>> Dtran = [];
    public HashSet<string> Alphabet = [];

    public DFA() { }
    public DFA(Node tree, Dictionary<int, string> indexedStates,
       Dictionary<int, HashSet<int>> treeFollowpos)
    {
        foreach (var s in indexedStates)
        {
            Alphabet.Add(s.Value);
        }
        int i = 0;
        Dstates.Add(new (tree.firstpos, i++));
        while (Dstates.Any(s => s.mark == false))
        {
            var state = Dstates.First(s => s.mark == false);
            state.mark = true;
            foreach (var symbol in Alphabet)
            {
                var fins = indexedStates.Where(ins => ins.Value
                   == symbol).Select(ins => ins.Key).ToHashSet();
                HashSet<int> U = [];
                var flag = false;
                foreach (var p in state.states)
                {
                    if (!fins.Contains(p))
                        continue;
                    if (symbol == "#")
```

```
                {
                    finishStates.Add(state.index);
                    break;
                }
                var ps = treeFollowpos[p];
                U = [.. U, .. ps];
            }
            if (flag) continue;
            DFAState newState = new(U, i);
            if (U.Count == 0)
                continue;
            var oldState = Dstates.FirstOrDefault(s => s ==
               newState);
            if (oldState is null)
            {
                Dstates.Add(newState);
                i++;
            }
            else
            {
                newState = oldState;
            }

            if (!Dtran.ContainsKey(state.index))
                Dtran[state.index] = [];
            Dtran[state.index][symbol] = newState.index;

        }
    }
}

public bool Accept(string data)
{
    var curState = startState;
    for (int i = 0; i < data.Length; i++)
    {
        Console.Write("|-{0}", data[i..]);
        int toState = Dtran.GetValueOrDefault(curState,
           []).GetValueOrDefault(data[i].ToString(), -1);
        if (toState == -1)
        {
```

```
                Console.WriteLine("|-no");
                return false;
            }
            curState = toState;
        }
        if (!finishStates.Contains(curState))
        {
            Console.WriteLine("|-no");
            return false;
        }
        Console.WriteLine("|-yes");
        return true;
    }
}

public class DFAState(HashSet<int> states, int index)
{
    public int index = index;
    public HashSet<int> states = states;
    public bool mark = false;

    public static bool operator ==(DFAState a, DFAState b)
    {
        if (a.states.Count != b.states.Count)
            return false;
        foreach(var s in a.states)
            if (!b.states.Contains(s))
                return false;
        return true;
    }

    public static bool operator !=(DFAState a, DFAState b)
    {
        if (a.states.Count == b.states.Count)
            return false;
        foreach (var s in a.states)
            if (!b.states.Contains(s))
                return true;
        return false;
    }
}
```

Листинг 3 – "Класс минимального ДКА"

```
public class MinDFA : DFA
{
    public MinDFA(DFA dfa)
    {
        Dictionary<int, Dictionary<string, HashSet<int>>>
           reverseDtran = [];
        reverseDtran[-1] = [];
        foreach (var state in dfa.Dstates)
        {
            foreach (var symbol in dfa.Alphabet)
            {
                if (dfa.Dtran.TryGetValue(state.index, out var
                   value) && value.TryGetValue(symbol, out int
                   fStateI))
                {
                    if (!reverseDtran.ContainsKey(fStateI))
                        reverseDtran[fStateI] = [];
                    if
                       (!reverseDtran[fStateI].ContainsKey(symbol))
                        reverseDtran[fStateI][symbol] = [];
                    reverseDtran[fStateI][symbol].Add(state.index);
                }
                else
                {
                    if (!reverseDtran[-1].ContainsKey(symbol))
                        reverseDtran[-1][symbol] = [];
                    reverseDtran[-1][symbol].Add(state.index);
                }
            }
        }
        foreach (var sym in dfa.Alphabet)
        {
            if (!reverseDtran[-1].ContainsKey(sym))
                reverseDtran[-1][sym] = [];
            reverseDtran[-1][sym].Add(-1);
        }

        List<DFAState> states = [new([], -1), .. dfa.Dstates];
        Dictionary<int, Dictionary<int, bool>> marked = [];
        foreach (var statei in states)
```

```
{
    marked[statei.index] = [];
    foreach (var statej in states)
    {
        marked[statei.index][statej.index] = false;
    }
}
Queue<KeyValuePair<int, int>> queue = [];

foreach (var stateI in states)
{
    foreach (var stateJ in states)
    {
        if (!marked[stateI.index][stateJ.index] &&
           dfa.finishStates.Contains(stateI.index) !=
           dfa.finishStates.Contains(stateJ.index))
        {
            marked[stateI.index][stateJ.index] = true;
            marked[stateJ.index][stateI.index] = true;
            queue.Enqueue(new (stateI.index,
               stateJ.index));
        }
    }
}

while (queue.Count != 0)
{
    var pair = queue.Dequeue();
    var u = pair.Key;
    var v = pair.Value;

    foreach (var c in dfa.Alphabet)
    {
        if (reverseDtran.TryGetValue(u, out var trans1)
           && trans1.TryGetValue(c, out var rSet))
        {
            foreach (var r in rSet)
            {
                if (reverseDtran.TryGetValue(v, out var
                   trans2) && trans2.TryGetValue(c, out
                   var sSet))
```

```
                {
                    foreach (var s in sSet)
                    {
                        if (!marked[r][s])
                        {
                            marked[r][s] = true;
                            marked[s][r] = true;
                            queue.Enqueue(new(r, s));
                        }
                    }
                }
            }
        }
    }
}


int newIndex = 0;
if (marked[-1].Values.Where(v => !v).Count() == 1)
    marked[-1][-1] = true;
foreach (var keyI in marked.Keys)
{
    HashSet<int> U = [];
    foreach (var keyJ in marked[keyI].Keys)
    {
        if (!marked[keyI][keyJ])
        {
            U.Add(keyJ);
        }
    }
    DFAState newState = new(U, newIndex);
    if (U.Count == 0)
        continue;
    var oldState = Dstates.FirstOrDefault(s => s ==
       newState);
    if (oldState is null)
    {
        Dstates.Add(newState);
        newIndex++;
    }
    else
```

```
        {
            newState = oldState;
        }
    }

    foreach (var state in Dstates)
    {
        if (state.states.Contains(dfa.startState))
            startState = state.index;
        foreach (var s in state.states)
        {
            if (dfa.finishStates.Contains(s))
                finishStates.Add(state.index);
        }
    }

    foreach (var state in Dstates)
    {
        foreach (var oldState in state.states)
        {
            if (reverseDtran.TryGetValue(oldState, out var
               tranSymbol))
            {
                foreach (var statesFrom in tranSymbol) {
                    foreach (var s in statesFrom.Value)
                    {
                        var newState =
                           Dstates.FirstOrDefault(ns =>
                           ns.states.Contains(s));
                        if (newState is not null)
                        {
                            if
                               (!Dtran.ContainsKey(newState.index)
                                Dtran[newState.index] = [];
                            Dtran[newState.index][statesFrom.Key]
                               = state.index;
                        }
                    }
                }
            }
        }
```

```
            }
        }
    }
```