Московский авиационный институт
(национальный исследовательский университет)

Факультет «Прикладная математика и физика»

Лабораторные работы по курсу «Системное программное обеспечение»

1. Спроектировать грамматику по заданному языку L
2. Спроектировать конечный автомат, составить диаграмму переходов КА и реализовать
3. Определить свойства КА. Построить НДКА. Реализовать преобразование НДКА в ДКА.

4. Устранить из КС-грамматики бесполезные символы и ε–правила
5 Устранить из КС-грамматики цепные правила и устранить левую рекурсию
6 Определить форму КС-грамматики и сделать ее приведение

7. Спроектировать МП автомат для приведенной КС-грамматики
8. Реализовать МП автомат для приведенной КС-грамматики

9. Для LL(1) анализатора построить управляющую таблицу M
10. Аналитически написать такты работы LL(1) анализатора для выведенной цепочки.
11. Реализовать управляющую таблицу M для LL(1) анализатора.

12. Построить замыкание множества ситуаций для пополненной LR(1) грамматики.
13. Определить функцию перехода g(x)
14. Построить каноническую форму множества ситуаций.
15. Построить управляющую таблицу для функции перехода g(x) и действий f(u).
16. Реализовать LR(1)-анализатор по управляющей таблице (g,f) для LR(1) грамматики.

Студент: Хренов Геннадий
Группа: 08-207Б

Руководитель: Семёнов А. С.

Оценка:
Дата:

Москва 2020

1.Спроектировать грамматику по заданному языку L:

Постановка задачи: L = L(G) = L(КА)

1.1. Задан бесконечный регулярный язык

$L = \{(0+1)(01)^{*}+\omega_1 \mid \omega_1\{0,1\}^{+}\}$

1.2. Преобразовать бесконечный регулярный язык(L) в конечный язык(L_1), цепочки символов которого являются подмножество цепочек символов бесконечного языка.

$L_1 = \{(0+1)+\omega_1 \mid \omega_1\{0,1, 00, 01, 10, 11\}\}$

1.3. Сгенерировать цепочки символов по языку L_1.

Лемма о накачке

(1)Для регулярного языка $L = \{(0+1)(01)^{*}+\omega_1 \mid \omega_1\{0,1\}^{+}\}$

(2)существует целое $(\exists p \geq 1 = 8$ такое что

(3)для $(0+1)+01 \in L((|(0+1)+01| \geq 8) \Rightarrow$

(4) существует $(\exists x = (0+1)+, y = 01, z = \varepsilon \in \Sigma$ такое что$(w=xyz \Rightarrow$

1.($|y| \geq 1$, цикл y должен быть накачан хотя бы длиной 1 и

2. $|xy| \leq p$, цикл должен быть в пределах первых p символов и

3. для всех $i \geq 0, (xy^{i}z \in L))))))))$, на x и z ограничений не накладывается.

$(0+1)+\omega_1 \Rightarrow (0+1)+0$

$(0+1)+\omega_1 \Rightarrow (0+1)+1$

$(0+1)+\omega_1 \Rightarrow (0+1)+01$

$(0+1)+\omega_1 \Rightarrow (0+1)+10$

$(0+1)+\omega_1 \Rightarrow (0+1)+11$

1.4. Спроектировать грамматику для языка L_1.

$S_0 \rightarrow (A$

$A \rightarrow 0B$

$B \rightarrow +C$

$C \rightarrow 1D$

$D \rightarrow)E$

$E \rightarrow +F$

$F \rightarrow 0$

$F \rightarrow 1$

$F \rightarrow 0G$

$F \rightarrow 1G$

$G \rightarrow 0$

$G \rightarrow 1$

$S_0 \Rightarrow (A \Rightarrow (0B \Rightarrow (0+C \Rightarrow (0+1D \Rightarrow (0+1)E \Rightarrow (0+1)+F \Rightarrow (0+1)+0$

S_0 => (A => (0B => (0+C => (0+1D => (0+1)E => (0+1)+F => (0+1)+1
S_0 => (A => (0B => (0+C => (0+1D => (0+1)E => (0+1)+F => (0+1)+0G => (0+1)+01
S_0 => (A => (0B => (0+C => (0+1D => (0+1)E => (0+1)+F => (0+1)+1G => (0+1)+10
S_0 => (A => (0B => (0+C => (0+1D => (0+1)E => (0+1)+F => (0+1)+1G => (0+1)+11

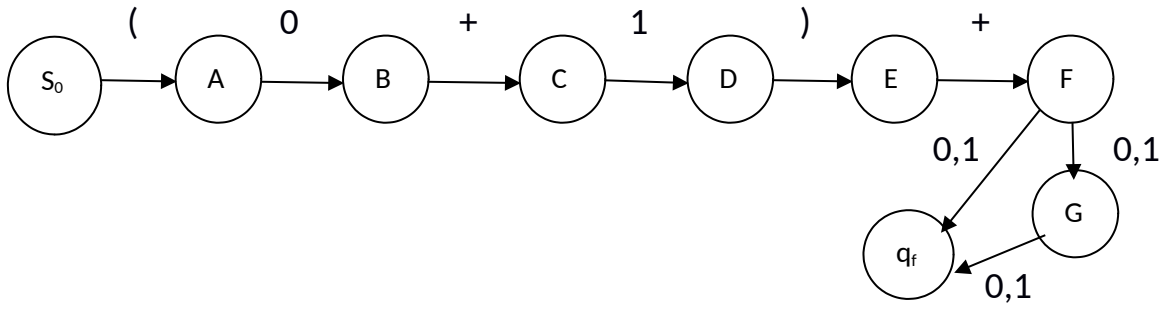
$L_1(G)$ = {(0+1)+0, (0+1)+1, (0+1)+01, (0+1)+10, (0+1)+11}

Множества L_1 и $L_1(G)$ совпадают, следовательно, данные языки совпадают и грамматика построена правильно.

2.Спроектировать конечный автомат, составить диаграмму переходов КА и реализовать

2.1. Построить диаграмму переходов и таблицу переходов по грамматике (1.3).

	0	1	+	(	)
S_0				{A}	
A	{B}				
B			{C}		
C		{D}			
D					{E}
E			{F}		
F	{G, q_f}	{G, q_f}			
G	{q_f}	{q_f}			
q_f					



KA = (S_0,A,B,C,D,E,F,qf), (0,1,+,(,),S_0,q_f,δ)
δ(S_0, `(`) = {A}
δ(A, 0) = {B}
δ(B, +) = {C}
δ(C, 1) = {D}
δ(D, `)`) = {E}

δ(E, +) = {F}
δ(F, 0) = { q_f}
δ(F, 1) = { q_f}
δ(F, 0) = {G}
δ(F, 1) = {G}
δ(G, 0) = {q_f}
δ(G, 1) = {q_f}

2.2. Реализовать конечный автомат по диаграмме переходов.
реализация представлена в выданном фреймворке:

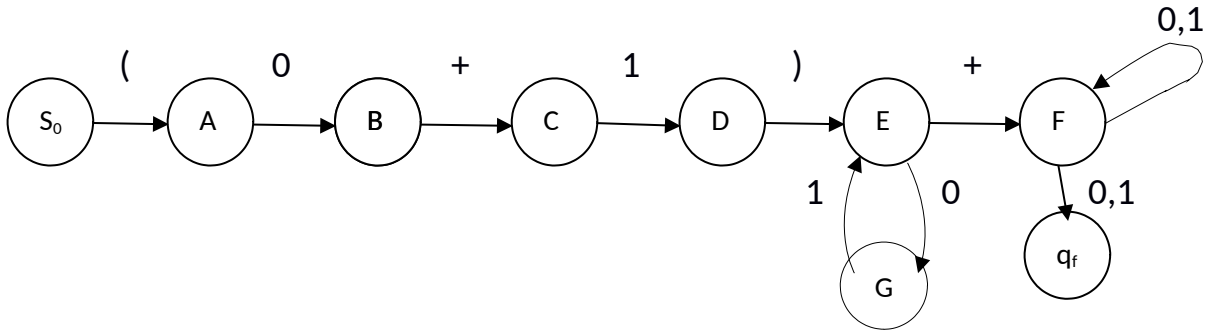
```
class myAutomate : Automate
    {
        public myAutomate(ArrayList Q, ArrayList Sigma, ArrayList F, string
q0) :
            base(Q, Sigma, F, q0) { }

        public myAutomate() : base() { }

        public void Execute(string chineSymbol)
        {
            string currState = this.Q0;
            int flag = 0;
            int i = 0;
            for (; i < chineSymbol.Length; i++)
            {
                flag = 0;
                foreach (DeltaQSigma d in this.DeltaList)
                {
                    if (d.leftNoTerm == currState && d.leftTerm ==
chineSymbol.Substring(i, 1))
                    {
                        currState = d.RightNoTerm;
                        flag = 1;
                        break;
                    }
                }
                if (flag == 0) break;
            } // end for

            Console.WriteLine("Length: " + chineSymbol.Length);
            Console.WriteLine(" i :" + i.ToString());
            Debug("curr", currState);
            if (this.F.Contains(currState) && i == chineSymbol.Length)
                Console.WriteLine("chineSymbol belongs to language");
            else
                Console.WriteLine("chineSymbol doesn't belong to language");
        } // end Execute
    } // KAutomate
```

3.Определить свойства КА. Построить НДКА. Реализовать преобразование НДКА в ДКА.

3.1. Сгенерировать цепочку символов по заданному языку L.

$$(0+1)(01)^*+\omega_1 \Rightarrow (0+1)01+\omega_1 \Rightarrow (0+1)01+0$$

3.2. По заданному языку построить НДКА. Представить в виде диаграмм.



НДКА= (S_0,A,B,C,D,E,F,qf), (0,1,+,(,),S0,q_f,δ)

δ(S_0, `(`) = {A}
δ(A, 0) = {B}
δ(B, +) = {C}
δ(C, 1) = {D}
δ(D, `)`) = {E}
δ(E, 0) = {G}
δ(G, 1) = {E}
δ(E, +) = {F}
δ(F, 0) = {F}
δ(F, 1) = {F}
δ(F, 0) = {q_f}
δ(F, 1) = {q_f}

3.3. Реализовать преобразование НДКА в ДКА.

```
public void BuildDeltaDKAutomate(myAutomate ndka)
        {
            this.Sigma = ndka.Sigma;
            this.DeltaList = ndka.DeltaList;
            ArrayList currState = EpsClosure(new ArrayList() { ndka.Q0 });

            config.Add(SetName(currState));
            Dtran(currState);
            this.Q = config;
            this.Q0 = this.Q[0].ToString();
            this.DeltaList = DeltaD;
            this.F = getF(config, ndka.F);
        }
```

4. Устранить из КС-грамматики бесполезные символы и ε-правила

4.1 Задана КС-грамматика: G =(T, V, P, S), где
T = {a, b}, V = {S, A, B, C}, P = {S → AB, S → SC, A→ BB, A→ Ab, A→ a, B → b, C→ Ca, C→ b}

4.2. Устранить бесполезные символы, ε-правила.

Для удаления бесполезных символов, сначала исключим непроизводящие терминалы, а затем недостижимые символы :

а)Если все символы цепочки из правой части правила вывода являются производящими, то нетерминал в левой части правила вывода также должен быть производящим:

Vp = { S,A,B,C} – непроизводящих нет

$G = G_1(T, Vp, P, S)$

б) Если нетерминал в левой части правила грамматика является достижимым, то достижимы и все символы правой части этого правила:

Vr = {S,A,B,C} – все символы достижимы

$G = G_2(T,Vr,B,C)$

Переходим к удалению ε- правил:

Алгоритм устранения ε–правил в КС- грамматике основан на использовании множества укорачивающих нетерминалов.

а) находим ε- правила

б) определяем множество неукорачивающихся символов

в) Для каждого правила вида $A \to \alpha 0 B 1 \alpha 1 B 2 \alpha 2 \ldots B k \alpha k$ (где αi — последовательности из терминалов и нетерминалов, Bj — ε-порождающие нетерминалы) добавить в P' все возможные варианты правил, в которых либо присутствует, либо удалён каждый из нетерминалов $Bj(1 \leqslant j \leqslant k)$

г)удаляем ε- правила

д) проверяем, есть ли порождение ε из S_0, в случае необходимости добавляем $S' \to S_0|\varepsilon$

В нашем случае V1 = { S,A,B,C} – ε-правил нет

4.3. Реализация алгоритма.

```
public myGrammar unUsefulDelete()
        {
            Console.WriteLine("\t\tDeleting unuseful symbols");
            Console.WriteLine("Executing: ");
            ArrayList Vp = new ArrayList();
            ArrayList Vr = new ArrayList();
            Vr.Add(this.S0);
            ArrayList Pp = new ArrayList();
            ArrayList P1 = new ArrayList(this.Prules);
            bool flag = false, noadd = false;
            // Создааем множество порождающих символов и одновременно
непроизводящие правила
            do
            {
                flag = false;
                foreach (Prule p in P1)
                {
                    noadd = false;
                    //DebugPrule(p);
                    if (p.RightChain == null || p.RightChain.Contains(""))
                    {
                        Pp.Add(p);
                        if (!Vp.Contains(p.LeftNoTerm))
                        {
```

```
                        Vp.Add(p.LeftNoTerm);
                    }
                    P1.Remove(p);
                    flag = true;
                    break;
                }
                else
                {
                    foreach (string t in p.RightChain)
                    {
                        if (!this.T.Contains(t) && !Vp.Contains(t))
                        {
                            //Console.WriteLine(t);
                            noadd = true;
                            break;
                        }
                    }
                    if (!noadd)
                    {
                        Pp.Add(p);
                        if (!Vp.Contains(p.LeftNoTerm))
                        {
                            Vp.Add(p.LeftNoTerm);
                        }
                        P1.Remove(p);
                        flag = true;
                        break;
                    }
                }
            }
        } while (flag);

        Debug("Vp", Vp);
        P1.Clear();
        if (!Vp.Contains(this.S0))
        {
            return new myGrammar(new ArrayList(), new ArrayList(), new
ArrayList(), this.S0);
        }
        ArrayList T1 = new ArrayList();
        //Создаем множество достижимых символов
        do
        {
            flag = false;
            foreach (Prule p in Pp)
            {
                if (Vr.Contains(p.leftNoTerm))
                {
                    foreach (string t in p.RightChain)
                    {
                        if (!Vr.Contains(t))
                        {
                            Vr.Add(t);
                            //noadd = true;
                        }
                    }
                    P1.Add(p);
                    Pp.Remove(p);
                    flag = true;
                    break;
                }
            }
        } while (flag);
```

```
            Debug("Vr", Vr);
            Vp.Clear();
            // Обновляем множества терминалов и нетерминалов
            foreach (string t in Vr)
            {
                if (this.T.Contains(t))
                {
                    T1.Add(t);
                }
                else if (this.V.Contains(t))
                {
                    Vp.Add(t);
                }
            }
            Debug("T1", T1);
            Debug("V1", Vp);
            Console.WriteLine("\tUnuseful symbols have been deleted");
            return new myGrammar(T1, Vp, P1, this.S0);
        }


public myGrammar EpsDelete()
        {
            Console.WriteLine("\tDelete e-rules:");
            Console.WriteLine("Executing:");
            ArrayList Erule = new ArrayList();
            ArrayList Ps = new ArrayList(this.Prules);
            //ArrayList NoTerm = new ArrayList();
            Console.WriteLine("e-rules:");
            //находим множество e-правил
            foreach (Prule p in this.Prules)
            {
                if (p.RightChain.Contains(""))
                {
                    DebugPrule(p);
                    Erule.Add(p);
                    Ps.Remove(p);
                }
            }
            //определяем множество неукорачивающихся символов
            ArrayList NoTerms = new ArrayList();

            foreach (Prule p in Erule)
            {
                if (!NoTerms.Contains(p.LeftNoTerm))
                {
                    NoTerms.Add(p.LeftNoTerm);
                }
            }
            bool flag = false, noadd = false;
            do
            {
                flag = false;
                foreach (Prule p in Ps)
                {
                    noadd = false;
                    //DebugPrule(p);
                    foreach (string t in p.RightChain)
                    {
                        if (!NoTerms.Contains(t))
                        {
                            noadd = true;
                            break;
```

```
                    }
                }
                if (!noadd)
                {
                    if (!NoTerms.Contains(p.LeftNoTerm))
                    {
                        NoTerms.Add(p.LeftNoTerm);
                    }
                    flag = true;
                    Ps.Remove(p);
                    break;
                }
            }
        } while (flag);
        Debug("NoShortNoTerms", NoTerms);
        Ps.Clear();
        //string s;
        //Удаляем e-правила и создаем новые в соответствии с алгоритмом
        foreach (Prule p in this.Prules)
        {
            if (Erule.Contains(p))
            {
                continue;
            }
            Ps.Add(p);
            for (int i = 0; i < p.RightChain.Count; ++i)
            {
                string t = p.RightChain[i].ToString();
                if (NoTerms.Contains(t))
                {
                    //s = t;
                    ArrayList NR = new ArrayList(p.RightChain);
                    NR.RemoveAt(i);
                    Ps.Add(new Prule(p.LeftNoTerm, NR));
                }
            }
        }
        //проверяем есть ли порождение e из нач символа
        if (NoTerms.Contains(this.S0))
        {
            ArrayList V1 = new ArrayList(this.V);
            V1.Add("S1");
            Ps.Add(new Prule("S1", new ArrayList() { this.S0 }));
            Ps.Add(new Prule("S1", new ArrayList() { "" }));
            Debug("V1", V1);
            Console.WriteLine("\te-rules have been deleted!");
            return new myGrammar(this.T, V1, Ps, "S1");
        }
        else
        {
            Debug("V1:", this.V);
            Console.WriteLine("\te-rules have benn deleted!");
            return new myGrammar(this.T, this.V, Ps, this.S0);
        }
    }
```

5. Устранить из КС-грамматики цепные правила и устранить левую рекурсию

5.1. Устранение цепных правил.

Сначала находим цепные правила, т. е. правила вида A→ B, где A и B - нетерминалы: (у нас таких нет)

Затем для правил, в правых частях которых есть нетерминал, для которого существует цепное правило(где он стоит слева), заменяем правую часть исходного правила на правую часть цепного правила.

5.2. Устранение левой рекурсии.

Для начала преобразуем G так, чтобы в правиле Ai → α, цепочка α начиналась либо с терминала, либо с такого Aj , что j > i.

V = { S,A,B,C}

P = {S → AB, S → SC, A→ BB, A→ Ab, A→ a, B → b, C→ Ca, C→ b}

Затем пусть множество Ai правил – это Ai → Aiα1 |…| Aiαm | β1 |…| βp, где ни одна цепочка βj не начинается с Ak, если k ≤ i. Заменим Ai - правила правилами:

Ai → β1 |…| βp | β1Ai’ |…| βpAi’

A’i → α1 |…| αm | α1Ai’ |…| αmAi’

где Ai’ – новый символ. Правые части всех Ai - правил начинаются теперь с терминала или с Ak для некоторого k > i.

В нашем случае:

P^1= {S → AB, S -> ABS’, S’ → C, S’ -> CS’, A→ BB, A -> BBA’, A’→ b, A’ -> bA’, A→ a, A -> aA’, B → b, C’ → a, C’ -> aC’, C→ b, C -> bC’}

5.3. Реализация алгоритма.

```
public myGrammar LeftRecursDelete()
        {
            Console.WriteLine("\tLeft Recursion delete:");
            Console.WriteLine("Executing: ");
            ArrayList P = new ArrayList();
            ArrayList V1 = new ArrayList(this.V);
            ArrayList Vr = new ArrayList();
            //ищем рекурсивные правила
            Console.WriteLine("Rules with Recursion:");
            foreach (Prule p in this.Prules)
            {
                if (p.LeftNoTerm == p.RightChain[0].ToString())
                {
                    DebugPrule(p);
                    if (!Vr.Contains(p.LeftNoTerm))
                    {
                        Vr.Add(p.LeftNoTerm);
                    }
                }
            }
            foreach (Prule p in this.Prules)
            {
                if (!Vr.Contains(p.LeftNoTerm))
                {
                    P.Add(p);
                }
            }
```

```
//преобразуем их в новые без левой рекурсии
ArrayList v_struct_ar = new ArrayList();

foreach (string v in Vr)
{
    V_struct v_struct;
    v_struct.alpha = new ArrayList();
    v_struct.betta = new ArrayList();
    v_struct.V = v;
    foreach (Prule r in this.Prules)
    {
        if (v == r.LeftNoTerm)
        {
            if (r.RightChain[0].ToString() == v)
            {
                ArrayList alpha_help = new ArrayList();
                for (int i = 1; i < r.RightChain.Count; i++)
                {
                    alpha_help.Add(r.RightChain[i]);
                }
                if (alpha_help.Count > 0)
                    v_struct.alpha.Add(alpha_help);
            }
            else
            {
                if (r.RightChain.Count > 0)
                    v_struct.betta.Add(r.RightChain);
            }
        }
    }
    v_struct_ar.Add(v_struct);
}

foreach (V_struct v_struct in v_struct_ar)
{
    string new_v = v_struct.V + "'";
    V1.Add(new_v);

    foreach (ArrayList betta_help in v_struct.betta)
    {
        P.Add(new Prule(v_struct.V, betta_help));
        ArrayList betta_pravila = new ArrayList();
        for (int i = 0; i < betta_help.Count; i++)
        {
            betta_pravila.Add(betta_help[i]);
        }
        betta_pravila.Add(new_v);
        P.Add(new Prule(v_struct.V, betta_pravila));
    }

    foreach (ArrayList alpha_help in v_struct.alpha)
    {
        P.Add(new Prule(new_v, alpha_help));
        ArrayList alpha_pravila = new ArrayList();
        for (int i = 0; i < alpha_help.Count; i++)
        {
            alpha_pravila.Add(alpha_help[i]);
        }
        alpha_pravila.Add(new_v);
        P.Add(new Prule(new_v, alpha_pravila));
    }
}
Debug("V1", V1);
Console.WriteLine("\tLeft Recursion have been deleted!");
```

```
            return new myGrammar(this.T, V1, P, this.S0);
        }
```

6. Определить форму КС-грамматики и сделать ее приведение

6.1. Задана следующая грамматика: G =(T, V, P, S), где

T = {i, + , -,(,)}, V = {S, F, L}, P = {S→ (F+L), S→ F, F→ - L, F→ i, L→F}

6.2. Эта же грамматика в приведённом виде:

Удалить:

1) ε- правила
2) цепных
3) бесполезных
4) левой рекурсии

T = {i, + , -,(,)}, V = {S, F, L}, P = {S → i, S→ (F+L), F → i, L → i, F → - L, S → -L, L → -L}

КС грамматика G = (T, V, P, S) называется грамматикой в нормальной форме Грейбах, если в ней нет ε-правил и каждое правило из P отличное от S → ε, имеет вид A→ aα, где a ∈T, α ⊂V^*.

КС грамматика G = (T, V, P, S) называется грамматикой в нормальной форме Хомского, если каждое правло из P имеет один из следующих видов:

1. A→ BC, где A, B, C∈V;
2. A→ a, где a∈T;
3. S→ ε, если ε∈L(G), причем S не встречается в правых частях правил.

Наша КС грамматика в ослабленной нормальной форме Грейбах.

7. Спроектировать МП автомат для приведенной КС-грамматики

Алгоритм построения:

1. Если A → α - правило грамматики G, то δ (q, ε, A) = (q,α).
2. δ (q, a, a) = {(q, ε)} для всех a∈Σ.

МП = ({q}, {i, + , -, (,)}, {i, + , -, (,), S, F, L}, δ, q_0, S, {q}), в котором функция переходов δ определяется следующим образом:

1. δ (q_0, ε, S) = {(q, (F+L)), (q, F)};
2. δ (q, ε, F) = {(q, -L), (q, i)};
3. δ (q, ε, L) = {(q, F)};
4. δ (q, a, a) = {(q, ε)} для всех a∈Σ={i, + , -,(,)}.

Последовательность тактов МП-автомата для цепочки (-i+i) :

$(q_0, (-i+i), S) \vdash_1 (q, (-i+i), (F+L)) \vdash_4 (q, -i+i), F+L)) \vdash_2 (q, -i+i), -L+L)) \vdash_4 (q, i+i), L+L)) \vdash_3 (q, i+i), F+L)) \vdash_2 (q, i+i), i+L)) \vdash_4 (q, +i), +L)) \vdash_4 (q, i), L)) \vdash_3 (q, i), F)) \vdash_2 (q, i), i)) \vdash_4 (q,),)) \vdash_4 (q, \varepsilon, \varepsilon).$

Отличие МП РМП автоматов заключается в том, что верхним элементом магазина является самый левый и самый правый символ цепочки соответственно.
Синтаксические анализаторы на основе МП автомата – нисходящие(предсказывающие), а на основе РМП автомата – восходящие.

8. Реализовать МП автомат для приведенной КС-грамматики

```
class myMp : Automate //МП = {}
    {
        // Q - множество состояний МП - автомата
        // Sigma - алфавит входных символов
        // DeltaList - правила перехода
        // Q0 - начальное состояние
        // F - множество конечных состояний
        public ArrayList Gamma = null; //алфавит магазинных символов

        public Stack Z = null;
        public string currState;
        public string ans1 = "";
        public string ans2 = "";

        // МП для КС-грамматик
        public myMp(myGrammar KCgrammar)
            : base(new ArrayList() { "q" }, KCgrammar.T, new ArrayList() { },
"q")
        {
            this.Gamma = new ArrayList();
            this.Z = new Stack();
            foreach (string v1 in KCgrammar.V) // магазинные символы
                Gamma.Add(v1);
            foreach (string t1 in KCgrammar.T)
                Gamma.Add(t1);
            Q0 = Q[0].ToString(); // начальное состояние
            Z.Push(KCgrammar.S0); // начальный символ в магазине
            F = new ArrayList(); // пустое множество заключительных состояний

            DeltaQSigmaGamma delta = null;

            foreach (string v1 in KCgrammar.V)
            { // сопоставление правил с отображениями
                ArrayList q1 = new ArrayList();
                ArrayList z1 = new ArrayList();
                foreach (Prule rule in KCgrammar.Prules)
                {
                    if (rule.leftNoTerm == v1)
                    {
                        Stack zb = new Stack();
                        ArrayList rr = new ArrayList(rule.rightChain);
                        rr.Reverse();
                        foreach (string s in rr)
                            zb.Push(s);
                        z1.Add(zb);
```

```
                        q1.Add(Q0);
                    }
                }
                delta = new DeltaQSigmaGamma(Q0, "e", v1, q1, z1);
                DeltaList.Add(delta);
            }

            foreach (string t1 in KCgrammar.T)
            {
                Stack e = new Stack();
                e.Push("e");
                delta = new DeltaQSigmaGamma(Q0, t1, t1, new ArrayList() { Q0
}, new ArrayList() { e });
                DeltaList.Add(delta);
            }
        }

        public virtual void addDeltaRule(string LeftQ, string LeftT, string
LeftZ, ArrayList RightQ, ArrayList RightZ)
        {
            DeltaList.Add(new DeltaQSigmaGamma(LeftQ, LeftT, LeftZ, RightQ,
RightZ));
        }

        public void addDeltaRule(string LeftQ, string LeftT, string LeftZ,
ArrayList RightQ, ArrayList RightZ, ArrayList RightSix)
        {
            DeltaList.Add(new DeltaQSigmaGammaSix(LeftQ, LeftT, LeftZ,
RightQ, RightZ, RightSix));
        }


    public virtual bool Execute_ (string str, int i, int Length) {
      //сразу нулевое правило брать
      DeltaQSigmaGamma delta = null;
      delta = (DeltaQSigmaGamma)this.DeltaList[0];
      currState = this.Q0;
//      int i = 0;  // sas!!
      int j = 0;
      str = str + "e"; // empty step вставить "" не получается, так как это
считается пустым символом,
                       //который не отображается в строке
      string s;
      delta.debug();
      for (; ; )
      {
        if (delta == null)
        {
          return false;
        }
        if (delta.LeftT != "")  // И В ВЕРШИНЕ СТЕКА ТЕРМИНАЛЬНЫЙ СИМВОЛ
LeftT!!!! пустой такт
        {
          for (; i < str.Length;) //модель считывающего устройства
          {
            if (Z.Peek().ToString() == str[i].ToString())
            {
              this.Z.Pop();
              currState = delta.RightQ.ToString();
              i++;
            } else return false;
            break;
          }
```

```
        } else if (delta.LeftT == "")  // И В ВЕРШИНЕ СТЕКА НЕ ТЕРМИНАЛЬНЫЙ
СИМВОЛ LeftT!!!!
        {
          //шаг 1 вытолкнуть из стека и занести в стек rightZ
          this.Z.Pop();
          s = arrToStr(delta.RightZ);
          for (j = s.Length - 1; j >= 0; j--) this.Z.Push(s[j]);
        }
        if (this.Z.Count != 0)
        {
          currState = arrToStr(delta.RightQ);

          this.debugDeltaRule("1", delta);
          //Execute_ (str,i, str.Length);

          delta = findDelta(currState, Z.Peek().ToString());
          delta.debug();
        } else if (str[i].ToString() == "e") return true;
        else return false;

      } // end for
        //проверка на терминал или нетерминал в вершине стека
        //изменение правила по верхушке стека
    } // end Execute_
```

9. Для LL(1) анализатора построить управляющую таблицу M

1. Если A → β – правило грамматики с номером i, то M(A, a) = (β, i) для всех a ≠ ε, принадлежащих множеству FIRST(A). (то есть найти все терминалы в начале цепочек, выводимых из данного нетерминала)
Если ε∈FIRST(β), то M(A, b) = (β, i) для всех b∈FOLLOW(A). (то есть найти все терминалы, которые могут следовать непосредственно за нашим нетерминалом в выводимых им цепочках)
2. M(a, a) = ВЫБРОС для всех a∈T.
3. M(⊥, ε) = ДОПУСК.
4. В остальных случаях M(X, a) = ОШИБКА для X(V ∪ T ∪ {⊥}) и a∈T{ε}

Возьмем грамматику G =(T, V, P, S), где
T = {i, + , -,(,)}, V = {S, F, L}, P = {p1: S→ (F+L), p2: S→ F, p3: F→ - L, p4: F→ i, p5: L→F}

Последовательно рассмотрим правила:

Правило грамматики	Множество	Значение M
p1: S→ (F+L)	FIRST((F+L)) = {(}	M(S,() = (F+L), 1
p2: S→ F	FIRST(F) = {-}	M(S,-) = F, 2
p3: F→ - L	FIRST(- L) = {-}	M(F,-) = -L, 3
p4: F→ i	FIRST(i) = {i}	M(F, i) = i, 4
p5: L→F	FIRST(F) = {-,i}	M(L,-) = M(L,i) = F, 5

затем построчно составляем таблицу:

	i	(	)	+	-	ε
S		(F+L), 1			F, 2	
F	i, 4				-L, 3	
L	F, 5				F, 5	
i	выброс					
(		выброс				
)			выброс			
+				выброс		
-					выброс	
⊥						допуск

10. Аналитически написать такты работы LL(1) анализатора для выведенной цепочки.

Рассмотрим работу алгоритма для цепочки символов (-i+i), порожденной LL(1) грамматикой.

Алгоритм находится в начальной конфигурации ((-i+i), S⊥, ε).

Текущая конфигурация	Значение M
((-i+i), S⊥, ε)	M(S,() = (F+L), 1
((-i+i), (F+L)⊥, 1)	M((,() = выброс
(-i+i), F+L)⊥, 1)	M(F,-) = -L, 3
(-i+i), -L+L)⊥, 13)	M(-,-) = выброс
(i+i), L+L)⊥, 13)	M(L,i) = F, 5
(i+i), F+L)⊥, 135)	M(F, i) = i, 4
(i+i), i+L)⊥, 1354)	M(i,i) = выброс
(+i), +L)⊥, 1354)	M(+,+) = выброс
(i), L)⊥, 1354)	M(L,i) = F, 5
(i), F)⊥, 13545)	M(F, i) = i, 4
(i), i)⊥, 135454)	M(i,i) = выброс
( ),)⊥, 135454)	M(),)) = выброс
(ε, ⊥, 135454)	M(ε,⊥) = допуск

Цепочка принадлежит языку, а последовательность номеров правил 135454 – её разбор.

11. Реализовать управляющую таблицу M для LL(k) анализатора.

```
class LLParser
    {
        private myGrammar G;
        private Stack<string> Stack;
        private DataTable Table;
```

```
        public string OutputConfigure = "";
        Hashtable FirstSet = new Hashtable();
        Hashtable FollowSet = new Hashtable();

        public LLParser(myGrammar grammar) //конструктор
        {
            G = grammar;
            Table = new DataTable("ManagerTable");
            Stack = new Stack<string>(); //создаем стек(магазин) для символов
            // Создадим таблицу синтаксического анализа для этой грамматики

            // Определим структуру таблицы
            Table.Columns.Add(new DataColumn("VT", typeof(String)));
            Console.WriteLine("Создадим таблицу. Сначала создадим по столбцу
для каждого из этих терминалов: ");
            foreach (string termSymbol in G.T)
            {
                Console.Write(termSymbol);
                Console.Write(", ");
                Table.Columns.Add(new DataColumn(termSymbol, typeof(Prule)));
            }
            Console.WriteLine("\nТакже создаем строку для Эпсилон");
            Table.Columns.Add(new DataColumn("EoI", typeof(Prule))); //
Epsilon
            ComputeFirstSets(grammar); // Вычисляем множество First
            ComputeFollowSets(grammar); // Вычисляем множество Follow
            for (int i = 0; i < grammar.V.Count; i++) // Рассмотрим
последовательно все нетерминалы
            {
                DataRow workRow = Table.NewRow(); //Новая строка
                workRow["VT"] = (string)grammar.V[i];

                Console.Write("Рассмотрим нетерминал ");
                Console.Write((grammar.V[i]));
                Console.Write("\n");

                List<Prule> rules = getRules((string)grammar.V[i]); //
Получим все правила, соответствующие текущему нетерминалу

                foreach (var rule in rules)
                {

                    List<string> currFirstSet =
First(rule.RightChain.Cast<string>().ToList());
                    foreach (var firstSymbol in currFirstSet)
                    {
                        if (firstSymbol != "")
                        {
                            // Добавить в таблицу
                            Console.Write("   Первый символ правила ");
                            Console.Write(rule.LeftNoTerm);
                            Console.Write(" -> ");
                            for (int j = 0; j < rule.RightChain.Count; j++)
                            {
                                Console.Write(rule.RightChain[j]);
                            }
                            Console.Write(" - ");
                            Console.WriteLine(firstSymbol);

                            workRow[firstSymbol] = rule;
                            Console.Write("   Это правило заносим в таблицу
на пересечении строки нетерминала ");
                            Console.Write(rule.LeftNoTerm);
                            Console.Write(" и столбца терминала ");
```

```
                            Console.WriteLine(firstSymbol);
                            Console.Write("\n");
                        }
                        else
                        {
                            List<string> currFollowSet =
Follow(rule.leftNoTerm);
                            foreach (var currFollowSymb in currFollowSet)
                            {
                                string currFollowSymbFix = (currFollowSymb ==
"") ? "EoI" : currFollowSymb;
                                workRow[currFollowSymbFix] = rule;
                            }
                        }
                    }
                }
                Table.Rows.Add(workRow);
            }
        }
```

12. Построить замыкание множества ситуаций для пополненной LR(1) грамматики.

Возьмем грамматику G =(T, V, P, S), где
T = {i, + , -,(,)}, V = {S, F, L}, P = {p1: S→ (F+L), p2: S→ F, p3: F→ - L, p4: F→ i, p5: L→F}

Способ 1. Построение LR(k) анализатора способом использования расширенного магазинного алфавита

Способ использования расширенного магазина состоит из трех шагов:
Шаг 1. Определение активных префиксов;
Шаг 2. Построение управляющей таблицы;
Шаг 3. Применение алгоритма «перенос-свёртка».

Правила пополненной грамматики:
p0: S' → S
p1: S→ (F+L)
p2: S→ F
p3: F→ - L
p4: F→ i
p5: L→F

Шаг 1. Определение активных префиксов
Символ переносится в магазин только в том случае, если он кодирует цепочку, «совместимую» с цепочкой, которая будет находиться в магазине после переноса.

Цепочка, кодируемая данным магазинным символом, совместима с цепочкой в магазине, если она является суффиксом магазинной цепочки после переноса данного символа.

Символ грамматики	Магазинный символ	кодируемая цепочка
S	S_0	⊥S
F	F_1 F_2 F_5	(F F F
L	L_1 L_3	(F+L -L
i	i_4	i
+	$+_1$	(F+
-	$-_3$	-
(	$(_1$	(
)	$)_1$	(F+L)

Шаг 2. Построение управляющей таблицы

Управление алгоритмом осуществляется при помощи двух функций, приведенных в таблице следующим образом:
1. Используя значение верхнего символа магазина и входного символа, алгоритм определяет значение функции действия: *ПЕРЕНОС или СВЕРТКА;*

2. При выполнении переноса определяется значение функции переходов, равное магазинному символу, который нужно втолкнуть в магазин;

3. Значение функции действия, равное СВЕРТКА(i), однозначно определяет этот шаг.

Работа алгоритма описывается в терминах конфигураций, представляющих собой тройки вида (αT, ax, π), где αT – цепочка магазинных символов (T-верхний символ магазина), ax – необработанная часть входной цепочки, π выход (строка из номеров правил), построенный к настоящему моменту времени.

Таблица состоит из двух подтаблиц – функции действия и функции переходов. Входным символам с ленты соответствуют столбцы таблицы, символам магазина – строки.

Функция действий f(u) определяется на множестве $(V_p \cup \{\perp\}) \times (T \cup \{\varepsilon\})$ по правилам:
1. Если A → β – правило грамматики с номером i, то для конфигурации (αT, ax, π), где T кодирует цепочку β, f(u) = C(i).

2. Если A → β – правило грамматики с номером i, то для конфигурации (αT, ax, π), где T кодирует некий префикс цепочки β (но не саму основу), f(u) = П.
3. Для конфигурации (S_0, ⊥, π), где S_0 кодирует цепочку ⊥S, f(u) = ДОПУСК.
4. В противном случае, f(u) = ОШИБКА.

Функция переходов g(X) определяется на множестве $(V_p \cup \{\perp\}) \times (V \cup T \cup \{\varepsilon\})$ по правилам:
1. Если для конфигурации (αT, ax, π) для входного символа a ϵ (V∪T) в таблице существует символ a_i, совместимый с цепочкой αTa, то g(X) = a_i.
2. В противном случае, g(X) = ОШИБКА.

	функция действий f(u)						функция переходов g(X)							
	i	+	-	(	)	⊥	S	F	L	i	+	-	(	)
S_0	П	П	П	П	П	Д		F_2		i_4		$-_3$	$(_1$	
F_1	П	П	П	П	П						$+_1$			
F_2	C(2)	C(2)	C(2)	C(2)	C(2)	C(2)								
F_5	C(5)	C(5)	C(5)	C(5)	C(5)	C(5)								
L_1	П	П	П	П	П									$)_1$
L_3	C(3)	C(3)	C(3)	C(3)	C(3)	C(3)								
i_4	C(4)	C(4)	C(4)	C(4)	C(4)	C(4)								
$+_1$	П	П	П	П	П			F_5	L_1	i_4		$-_3$		
$-_3$	П	П	П	П	П			F_5	L_3	i_4		$-_3$		
$(_1$	П	П	П	П	П			F_1		i_4		$-_3$		
$)_1$	C(1)	C(1)	C(1)	C(1)	C(1)	C(1)								
⊥	П	П	П	П	П	П	S_0	F_2		i_4		$-_3$	$(_1$	

Шаг 3. Применение алгоритма «перенос-свёртка».
Алгоритм.
1. j := 0.
2. j := j+1. Если j > n, то выдать сообщение об ошибке и перейти к шагу 5.
3. Определить цепочку u следующим образом:
• если k = 0, то u= **tj**;
• если k ≥ 1 и j + k – 1 ≤ n, то u= tj**tj+1…tj+k–1** – первые k-символов цепочки tjtj+1…tn;
• если k ≥ 1 и j + k – 1 > n, то u= tj**tj+1…tn** – остаток входной цепочки.

4. Применить функцию действия f из строки таблицы M, отмеченной верхним символом магазина T, к цепочке u:
f(u) = ПЕРЕНОС (П). Определить функцию перехода g(tj) из строки таблицы M, отмеченной символом T из верхушки магазина. Если g(tj) = T' и T' ϵ Vp ∪ {⊥}, то записать T' в магазин и перейти к шагу 2. Если g(tj)= ОШИБКА, то выдать сигнал об ошибке и перейти к шагу 5;

• f(u) = (СВЕРТКА, i) (С) и A→α – правило вывода с номером i грамматики G. Удалить из верхней части магазина |α| символов, в результате чего в верхушке магазина окажется символ T' ϵ Vp ∪{⊥}, и выдать номер правила i на входную ленту. Определить символ T"= g(A) из строки таблицы M, отмеченной символом T', записать его в магазин и перейти к шагу 3.
• f(u) = ОШИБКА (О). Выдать сообщение об ошибке и перейти к шагу 5.
• f(u) = ДОПУСК (Д). Объявить цепочку, записанную на входной ленте, правым разбором входной цепочки z.

5. Останов.

Рассмотрим последовательность тактов LR(k)-алгоритма при анализе входной цепочки : (-i+i)
(⊥, (-i+i)⊥, ε) |- (⊥($_1$, -i+i)⊥, ε) |- (⊥($_1$-$_3$, i+i)⊥, ε) |- (⊥($_1$-$_3$$i_4$, +i)⊥, ε) |- (⊥($_1$-$_3$$F_5$, +i)⊥, 4) |- (⊥($_1$-$_3$$L_3$, +i)⊥, 45) |- (⊥($_1$$F_1$, +i)⊥, 453) |- (⊥($_1$$F_1$+$_1$, i)⊥, 453) |- (⊥($_1$$F_1$+$_1$$i_4$,)⊥, 453) |- (⊥($_1$$F_1$+$_1$$i_4$,)⊥, 453) |- (⊥($_1$$F_1$+$_1$$F_5$,)⊥, 4534) |- (⊥($_1$$F_1$+$_1$$L_1$,)⊥, 45345) |- (⊥($_1$$F_1$+$_1$$L_1$)$_1$, ⊥, 45345) |- (⊥$S_0$, ⊥, 453451) |- ДОПУСК

Способ 2. Построение LR(k) анализатора способом грамматических вхождений

Способ использования грамматических вхождений состоит из четырёх шагов:
Шаг 1. Определение грамматических вхождений (см. алгоритм фреймворка);
Шаг 2. Построение конечного автомата из грамматических вхождений;
Шаг 3. Построение управляющей таблицы.
Шаг 4. Применение алгоритма «перенос-свёртка».

Построение:
Шаг 1.Определение грамматических вхождений
Грамматическое вхождение – это символы полного словаря грамматики, снабженные двумя индексами. Первый индекс i задает номер правила грамматики, в правую часть которого входит данный символ, а второй индекс j – номер позиции символа в этой правой части.

В приведённой выше грамматике с правилами:
p0: S' → S
p1: S→ (F+L)
p2: S→ F
p3: F→ -L
p4: F→ i
p5: L→ F

выделяем грамматические вхождения: S_{01}, $(_{11}$, F_{12}, $+_{13}$, L_{14}, $)_{15}$, F_{21}, $-_{31}$, L_{32}, i_{41}, F_{51}

Шаг 2. Построение конечного автомата из грамматических вхождений
12.2. Определить множество First для LR(1) грамматики
First = {(, -, i}
12.3. Определить множество LR(1) ситуаций

LR(1) ситуация – объект вида [A → α • β, a], где (A → αβ) ∈ P, • ∈ V ∪ T, a ∈ {$} ∪ T, где $ - конец строки
LR(1) ситуация допустима для активного префикса +, если существует вывод $S \Rightarrow_r {}^*\gamma Aw \Rightarrow_r \gamma\alpha\beta w$, где + = γα и либо a – первый символ w, либо w = ε и a = $
Ситуация допустима, если она допустима для какого-то алфавитного префикса.
Считая, что: α и β ∈ {ε}∪ T ∪ V:
I = {(S' → •S, $), {(S' → S•, $), (S→ •(F+L), $), (S→ (•F+L), $), (S→ (F•+L), $), (S→ (F+•L), $), (S→ (F+L•), $), (S→ (F+L)•, $), (S→ •F, $), (S→ F•, $), (F → •i, +), (F → i•, +), (F→ •-L, +), (F→ -•L, +), (F→ -L•, +), (L→ •F,)), (L→ F•,)), (F → •i, $), (F → i•, $)}

12.4. Построить замыкание closure(I) множества ситуаций

CLOSURE(I) строится по следующим правилам:
1. Включить в CLOSURE(I) все ситуации из I.
2. Если ситуация A → α •Bβ уже включена в CLOSURE(I) и B → γ - правило грамматики, то добавить в множество CLOSURE(I) ситуацию B → • γ при условии, что там ее еще нет.
3. Повторять правило 2, до тех пор, пока в CLOSURE(I) нельзя будет включить новую ситуацию.

В нашем случае CLOSURE(I) = {(S' → •S, $), {(S' → S•, $), (S→ •(F+L), $), (S→ (•F+L), $), (S→ (F•+L), $), (S→ (F+•L), $), (S→ (F+L•), $), (S→ (F+L)•, $), (S→ •F, $), (S→ F•, $), (F → •i, +), (F → i•, +), (F→ •-L, +), (F→ -•L, +), (F→ -L•, +), (L→ •F,)), (L→ F•,)), (F → •i, $), (F → i•, $)}

Например, CLOSURE({S' → S}) = {S' → •S, S→ •(F+L), S→ •F, F→ •-L, F→ •i}

13. Определить функцию перехода g(x)
13.1. Определить функцию перехода goto(I,x)

Функция GOTO(I,X) определяется как замыкание множества всех ситуаций [A→ αX·β], таких что [A→ α·Xβ] ϵ I

GOTO(I,+) = {S → (F+•L), L → •F, F → •-L, F → •i }

GOTO(I,-) = {F → -•L, L → •F, F →•-L, F → •i}
GOTO(I,i) = {F → i•}
GOTO(I,() = {S → (•F+L), F → •-L, F → •i}
GOTO(I,)) = {S → (F+L)•}
GOTO(I,S) = {S'→ S•}
GOTO(I,F) = {L → F•, S → (F•+L), S → F•}
GOTO(I,L) = {F →-L•, S → (F+L•)}

14. Построить каноническую форму множества ситуаций.

14.1. Построить каноническую форму множества ситуаций

Процесс построения канонической системы множеств *LR(0)* –ситуаций можно описать с помощью следующих действий:

1.$\varphi=\varnothing$

2. Включить в φ множество *A0 = CLOSURE([S'→ • S])*, которое в начале «не отмечено».

3. Если множество ситуаций *A,* входящее в систему , «не отмечено», то:

• отметить множество *A*;

• вычислить для каждого символа $X \in (V \cup \Sigma)$ значение $A' = GOTO(A,X);$

• если множество $A' \neq \varnothing$ и еще не включено в φ, то включить его в систему множеств как «неотмеченное» множество.

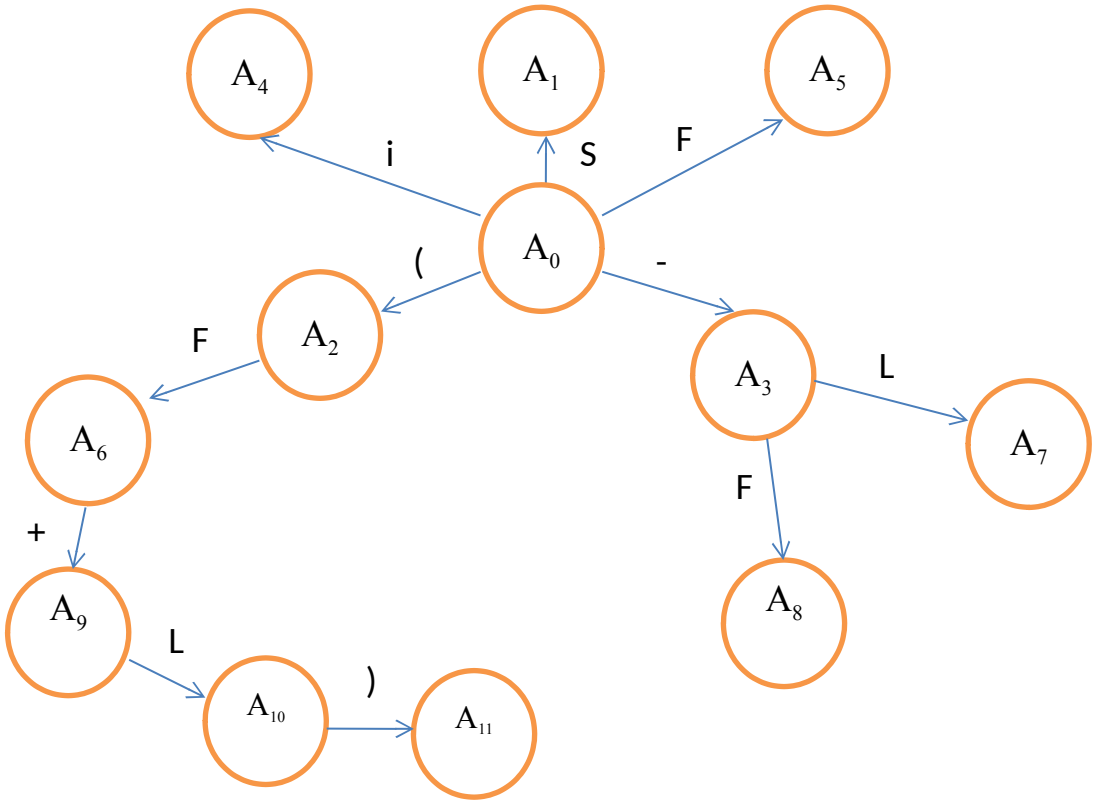
4. Повторять шаг 3, пока все множества ситуаций системы φ не будут отмечены.

Построение:

A_0 = CLOSURE(|S' → •S|)
A_1 = GOTO(A_0,*S*) = {S' → S•}
A_2 = GOTO(A_0,() = CLOSURE([*S*→(•F+L)]) = { *S* → (•F+L), F → •-L, F → •i}
A_3 = GOTO(A_0,-) = CLOSURE([F → -•L]) = { F → -•L, L → •F, F → •i, F → •-L}
A_4 = GOTO(A_0,i) = {F → i•}
A_5 = GOTO(A_0,F) = {S → F•}
Для всех X ∈ (V ∪ Σ) множества GOTO(A_1, X) пусты.
A_6 = GOTO(A_2,F) = {S → (F•+L)}
A = GOTO(A_2,-) = CLOSURE([F → -•L]) = A_3 - не добавляем
A_7 = GOTO(A_3,L) = { F → -L•}
A_8 = GOTO(A_3,F) = { L → F•}
A_9 = GOTO(A_6,+) = CLOSURE([S → (F+•L)]) = { S → (F+•L), L → •F, F → •i, F → •-L}
A_{10} = GOTO(A_9,L) = { S → (F+L•)}
A_{11} = GOTO(A_{10},)) = { S → (F+L)•}

A_0	$S' \rightarrow \bullet S$ $S \rightarrow \bullet(F+L)$ $S \rightarrow \bullet F$ $F \rightarrow \bullet\text{-}L$ $F \rightarrow \bullet i$
A_1	$S' \rightarrow S\bullet$
A_2	$S \rightarrow (\bullet F+L)$ $F \rightarrow \bullet\text{-}L$ $F \rightarrow \bullet i$
A_3	$F \rightarrow \text{-}\bullet L$ $L \rightarrow \bullet F$ $F \rightarrow \bullet i$ $F \rightarrow \bullet\text{-}L$
A_4	$F \rightarrow i\bullet$
A_5	$S \rightarrow F\bullet$
A_6	$S \rightarrow (F\bullet+L)$
A_7	$F \rightarrow \text{-}L\bullet$
A_8	$L \rightarrow F\bullet$
A_9	$S \rightarrow (F+\bullet L)$ $L \rightarrow \bullet F$ $F \rightarrow \bullet i$ $F \rightarrow \bullet\text{-}L$
A_{10}	$S \rightarrow (F+L\bullet)$
A_{11}	$S \rightarrow (F+L)\bullet$

14.2. Построить диаграмму переходов автомата



15. Построить управляющую таблицу для функции перехода g(x) и действий f(u).

Алгоритм:
1. Если f(α, aw) = ПЕРЕНОС, то входной символ переносится в верхушку магазина и читающая головка сдвигается на один символ вправо.
$(\alpha, aw, \pi) \vdash^{s} (\alpha a, w, \pi)$ для $\alpha \in (N \cup \Sigma \cup \{\perp\})^*$, $w \in (\Sigma \cup \{\perp\})^*$ и $\pi \in \{1, \ldots, p\}^*$.

2. Если f(αβ, w) = СВЕРТКА, g(αβ, w) = i и $A \rightarrow \beta$ – правило грамматики с номером i, то цепочка β заменяется левой частью правила с номером i, а его номер помещается в выходную ленту
$(\alpha\beta, w, \pi) \vdash^{r} (\alpha A, w, \pi i)$.
3. Если f(α, w) = ДОПУСК, то $(\alpha, w, \pi) \vdash^{s}$ ДОПУСК

	i	+	-	(	)	$	S	F	L
0	П(4)		П(3)	П(2)			1	5	
1						допуск			
2								6	
3								8	7
4	С(4)	С(4)	С(4)	С(4)	С(4)	С(4)			
5	С(2)	С(2)	С(2)	С(2)	С(2)	С(2)			
6		П(9)							
7	С(3)	С(3)	С(3)	С(3)	С(3)	С(3)			
8	С(5)	С(5)	С(5)	С(5)	С(5)	С(5)			
9									10
10					П(11)				
11	С(1)	С(1)	С(1)	С(1)	С(1)	С(1)			

Шаг 4**.** Применение алгоритма «перенос-свёртка».
Работа алгоритма описывается в терминах конфигураций, представляющих собой тройки вида (αT, ax, π), где αT – цепочка магазинных символов (T-верхний символ магазина), ax – необработанная часть входной цепочки, π – выход, построенный к настоящему моменту времени.

Рассмотрим последовательность тактов алгоритма при анализе входной цепочки (-i+i)
(⊥0, (-i+i)$, ε) |- (⊥0(2, -i+i)$, ε) |- (⊥0(2-3, i+i)$, ε) |- (⊥0(2-3i4, +i)$, ε) |- (⊥0(2-3F8, +i)$, 4) |- (⊥0(2-3L7, +i)$, 45) |- (⊥0(2F6, +i)$, 453) |- (⊥0(2F6+9, i)$, 453) |- (⊥0(2F6+9i4,)$, 453) |- (⊥0(2F6+9F8,)$, 4534) |- (⊥0(2F6+9L10,)$, 45345) |- (⊥0(2F6+9L10)11, $, 45345) |- (⊥0S1, $, 453451) |- ДОПУСК

16. Реализовать LR(1)-анализатор по управляющей таблице (g,f) для LR(1) грамматики.

Все реализации представлены в выданном фреймворке.