【Experiment name】

Chomsky grammar judgment type

【Purpose and Requirements of the Experiment】

Input: an arbitrary set of rules

Output: the type of the corresponding Chomsky grammar

(1) Grammar input should be simple

(2) Indicate which type of Chomsky grammar it is, and give the corresponding quadruple form: G = (VN, VT, P, S) .

Explanation: For the sake of simplicity, the o-type grammar class can be ignored

【Experimental principle】

1. Type 0 grammar (phrase grammar)

If for some grammar G, each rule in P has the form: u:: = v

Where u∈V+, v∈V*, the grammar G is called a type-0 grammar or a phrase grammar, abbreviated as PSG. The corresponding language of type 0 grammar or phrase structure grammar is called type 0 language or phrase structure language L0. Since this kind of grammar has no other restrictions, type 0 grammar is also called unrestricted grammar, and its corresponding language is called unrestricted language. Any type 0 language is recursively enumerable, so a type 0 language is also called a recursively enumerable set. This language can be recognized by a Turing machine.

2. Type 1 Grammars (Context Sensitive Grammars)

If for some grammar G, each rule in P has the form: xUy:: = xuy

Where U∈VN; u∈V+; x, y∈V*, the grammar G is called Type 1 grammar or context-sensitive grammar, also called context-sensitive grammar, abbreviated as CSG.

The U in the left part of the rule of Type 1 grammar and the u in the right part have the same upper x and lower y. When using this rule for derivation, to replace U with u must be preceded by x and followed by y. carried out, showing the context-sensitive features.

The language determined by Type 1 grammar is Type 1 language L1, and Type 1 language can be recognized by linear bounded automata.

3. Type 2 grammars (context-free grammars)

If for some grammar G, each rule in P has the following form:

U :: = u

Where U∈VN; u∈V+, the grammar G is called type 2 grammar or context-free grammar, abbreviated as CFG.

According to this rule, for a context-free grammar, when using this rule to deduce, it does not need to consider the context of the non-terminal U, and can always replace U with u, or reduce u to U, showing the characteristics of context-free.

The language determined by the Type 2 grammar is a Type 2 language L2, and the Type 2 language can be recognized by a non-deterministic pushdown automaton.

In general, grammars defining programming languages are context-free. Such as the C language is the case. Therefore, context-free grammars and corresponding languages have aroused people's greater interest and attention.

4. Type 3 grammars (regular grammars, linear grammars)

If for some grammar G, each rule in P has the following form:

U :: = T or U :: = WT

where T ∈ VT; U, W ∈ VN, then the grammar G is called a left-linear grammar.

If for some grammar G, each rule in P has the following form:

U::=T or U::=TW

Where T ∈ VT; U, W ∈ VN, the grammar G is called a right-linear grammar.

Left-linear grammar and right-linear grammar are collectively referred to as type 3 grammar or regular grammar, sometimes also called finite state grammar, abbreviated as RG.

By definition, when applying rules to a regular grammar, a single nonterminal can only be replaced by a single terminal, or by a single nonterminal plus a single terminal, or by a single terminal plus a single nonterminal .

The language determined by the Type 3 grammar is a Type 3 language L3, and the Type 3 language can be recognized by a definite finite state automaton.

In common programming languages, most lexical-related grammars belong to Type 3 grammars.

It can be seen that for the above four types of grammars, from type 0 to type 3, the production restrictions are getting stronger and stronger, and the latter type is a subset of the former type, while the function of the description language is getting weaker and weaker. The four types of grammar The relationship between the language and its expression can be expressed as : Type 0 Type ⊃1 Type ⊃2 Type ⊃3; namely L0 ⊃L1 ⊃L2 ⊃L

【Experimental content】

I wrote the relevant experimental code to realize the judgment of the Chomsky grammar of the input production. There are 5 possible situations in the judgment result, which are

1) Non-Chomsky grammar (irregular input, including illegal characters, etc.)

2) Type 0 grammar

3) Type 1 grammar

4) Type 2 grammar

5) Type 3 grammar

Among them, Type 3 grammar is further divided into left-linear grammar and right-linear grammar, which can be judged in time according to the input situation.

In order to realize this judgment procedure, I designed the following procedure flow:

1) Define the data used by the program as follows:

① Use Principle to store production expressions, which are divided into left type left and right type right .

②Using the character set V N defined by vector , VT stores non-terminal symbols and terminal symbols respectively according to the input of the production formula, and uses type = 1 or 2 to represent their types

```
class Principle
{
public :
    string left;
    string right;
    Principle( const char *, const char *);
};
vector < char > VN; // non-terminal character set
vector < char > VT; // Terminator set
vector < Principle > principle; // A collection of productions
```

```
int type[VSIZE]; // The type of each character , 1: non-terminal, 2: terminal

void init(); // data clear initialization
int get_type( char ); // Get character type
bool set_type( char , int ); // Set character type

int analysis_result(); // Get the type of input grammar
```

In the judgment of the C homsky grammar, it is mainly based on the experimental principle, and it is judged from type 0 to type 3, such as:

① If the input of illegal characters or does not satisfy the situation that a -> b consists of left and right formulas, it is considered not to be type 0

② If the length of the "left string" is less than the length of the "right string", it is not type 1

③If the "left string" does not satisfy the requirement that there is and only one non-terminal symbol, it is not type 2

④ If the "right string" exceeds 2 characters, or the right string is only non-terminal, it is not type 3

⑤ When ④ is satisfied, if the "right string" is only in the form of "terminal + non-terminal" or "terminal", it is "type 3 right linear grammar"; if the "right string" is only "non-terminal + terminal symbol" or "terminal symbol", it is "type 3 left-linear grammar". Otherwise, it is considered not to be "type 3".

【Experimental experience】

In this experiment, I carefully reviewed the judgment rules of the compilation principle Chomsky grammar.

When implementing Type 3 grammar, I thought that only "right linear grammar" was adopted as the definition of Type 3 grammar in the book, but during the experiment, I realized that "left linear grammar" is also part of Type 3 grammar. In realizing the judgment of left and right linear grammar, I found that when there is a situation like ″S ->aA A->aA A->Aa A->a ″, it does not belong to right linear grammar or left Linear grammar, then it is not a special case of Type 3 grammar. In order to achieve this judgment, I found that we do not need to judge whether it is possible to type 3 and then judge ″left and right linear″, but directly judge whether it is ″left and right by linear ″, because Type 3 grammar is only composed of these two, then the possibility of Type 3 grammar can be ruled out if it does not meet the two.

In this grammar judgment task, the judgment rules are relatively clear when understood, but the possible complexity of the input still aggravates the accidents of code errors. For this reason, I have considered various input situations as much as possible to improve the robustness of my own code.

Among them, Type 3 grammar is also a regular grammar, which satisfies the equivalent transformation relationship with DFA, NFA, and regular expressions, and they can realize equivalent transformations among them through corresponding rules.

Through experimental practice, I have improved my understanding of Chom sky grammar, and considered a variety of possibilities, changed and made up for the previous erroneous knowledge of its rules, and enhanced my coding ability. Experiment to lay the groundwork.

【Experiment code and results】

Experiment code:

Compile_1.cpp

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <string>
#include <vector>
```

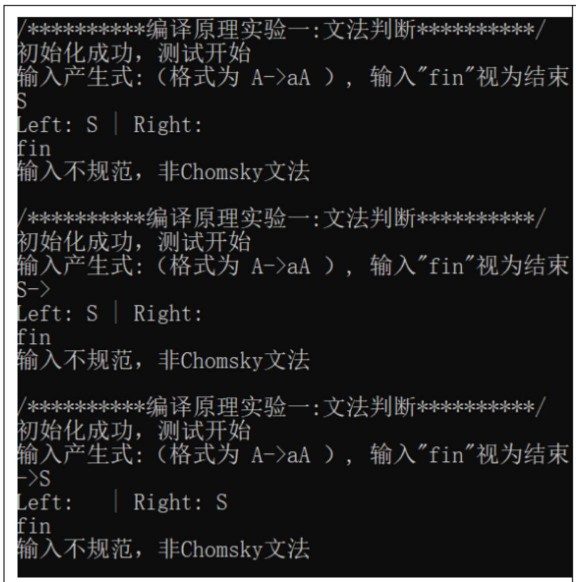
```
#include <cstring>

using namespace std;
const int VSIZE = 300;

class Principle
{
public:
    string left;
    string right;
    Principle( const char *, const char *);
};
vector < char > VN; // non-terminal character set
vector < char > VT; // Terminator set
vector < char > EP; // empty set
vector < Principle > principle; // A collection of productions
int type[VSIZE]; // The type of each character , 1: non-terminal, 2: terminal

void init(); // data clear initialization
int get_type( char ); // Get character type
bool set_type( char , int ); // Set character type
int analysis_result(); // Get the type of input grammar

int main()
{
    char buf[1000];
    //char** elements;
    while ( true )
    {
        init();  // data clear initialization
        puts( " Enter the production formula : (the format is A->aA ) , enter \"fin\" as the end " );
        while ( true )
        {
            gets_s(buf); //get string
            if (!strcmp(buf, "fin")) break;    //如为fin，strcmp()=0，!strcmp=1
            for (int i = 0; i < strlen(buf); i++) {
                if (isupper(buf[i])) {
                    char ch = buf[i];
                    if (get_type(ch)) continue; //return type[ch]
                    VN.push_back(ch);
                    set_type(ch, 1);  //1为非终结符
                }
                else if (islower(buf[i]) || isdigit(buf[i])) {  // || (buf[i] >= 48 && buf[i] <= 57)
```

```
                    char ch = buf[i];
                    if (get_type(ch)) continue;    //return type[ch]
                    VT.push_back(ch);
                    set_type(ch, 2);  //2为终结符
                }
                else if (buf[i] == '#') {  //Epsilon用#表示
                    char ch = buf[i];
                    if (get_type(ch)) continue;
                    EP.push_back(ch);
                    set_type(ch, 3);
                }
            }
            int i; int len = strlen(buf);
            for (i = 0; i < strlen(buf); i++) {     // input format -> , +3 from left to right
                if (buf[i] == '-' )
                {
                    buf[i] = 0;
                    i = i + 2;
                    break ;
                }
            }

            if (i<=2 || !strlen(buf)) {
                printf("Illegal input !\n");
                break ;
            }
            principle.push_back(Principle (buf, buf + i));  //buf : left string buf+i: right string
            printf("Left: %s\t| Right: %s\n", buf, buf + i);
        }

        int flag = analysis_result();

        switch (flag)
        {
        case -1:
            puts("Irregular input, non-Chomsky grammar");
            break ;
        case 0:
            puts("This grammar is type 0 grammar");
            break ;
        case 1:
            puts("This grammar is type 1 grammar");
            break ;
        case 2:
            puts( " This grammar is a Type 2 grammar " );
            break ;
        case 3:
            puts( " This grammar is type 3 grammar, left linear grammar " ); // Left linear grammar
            break ;
        case 4:
            puts( " This grammar is type 3 grammar, right linear grammar " ); // right linear grammar
            break ;
        }
        if (flag != -1)
        {
            printf( "\ nquad G=[Vn,Vt,P,S]\nVn:" );
            for (int i = 0; i < VN.size(); i++) {
                printf("%c ", VN.at(i));
            }
            printf("\nVt:");
            for (int i = 0; i < VT.size(); i++) {
                printf("%c ", VT.at(i));
            }
            printf("\nP: ");
            for (int i = 0; i < principle.size(); i++) {
                //printf("%c->%c ", principle.at(i).left, principle.at(i).right);
                cout << principle.at(i).left << "->" << principle. at(i). right << " " ;
            }
            printf( "\nS: %c\n" , VN. at(0));
            if (EP. size() == 1)
                printf( " The grammar contains epsilon\n" );
            else
                printf( " This grammar does not contain epsilon\n" );
        }
        printf( "\n" );
    }
    return 0;
}
// Start address of left and right strings
Principle ::Principle( const char * l , const char * r )
{
    left = l ;
    right = r ;
}
// Get character type
int get_type( char ch )
```

```
{
    return type[ ch ];
}
// Set character type
bool set_type( char ch , int x )
{
    type[ ch ] = x ;
    return true ;
}

// Judge whether the string input is standardized (whether it contains unknown characters,
whether it conforms to the style of a->b )
bool inputError( const string & s )
{
    for ( int i = 0; i < s . length(); i++)
        if (!get_type( s [ i ] )) return true ;     // unknown character:
non-uppercase, lowercase or number
    return false ;
}

// Determine whether it is a type 0 grammar
bool isZero()
{
    if (principle. size() == 0)
        return false ;
    for ( int i = 0; i < principle. size(); i++) {
        if (inputError(principle [ i ] .left)) return false ;    // left string
        else if (inputError(principle [ i ] .right)) return false ;  // right string
        else if (principle [ i ] . left. length() == 0) return false ;    // left
string is empty
        else if (principle [ i ] .right.length() == 0) return false ;     // right
string is empty

    }
    return true ;
}

// Check if a type 0 grammar is a type 1 grammar
bool isOne()
{
    for ( int i = 0; i < principle. size(); i++) {
        if (principle [ i ] .left.length() > principle [ i ] .right.length())  // If
"left string" <= "right string", it is type 1
            return false ;
    }
    return true ;
}

// Check if a type 1 grammar is a type 2 grammar
bool isTwo()
{
    for ( int i = 0; i < principle. size(); i++)
    {
        string left = principle [ i ] .left;
        if (left. size() != 1) return false ;        // "left string" has only 1 element
        if (get_type(left [ 0 ] ) != 1) return false ;   // For example, the first letter
of "left string" is lowercase
    }
    return true ;
}

// Determine whether a type 2 grammar is a left-linear grammar ( type 3 )
bool isLeftThree()
{
    for ( int i = 0; i < principle. size(); i++)
    {
        string right = principle [ i ] .right;
        if (right. length()==1)
            if (get_type(right [ 0 ] ) == 1)
                return false ;                 // If the first letter of the right string
is a terminal
        for ( int j = 1; j < right. length(); j++)
        {
            if (get_type(right [ j ] ) ==1 ) return false ;

        }
    }
    return true ;
}

// Determine whether a type 2 grammar is a right linear grammar ( type 3 )
bool isRightThree()
{
    for (int i = 0; i < principle.size(); i++)
    {
        string right = principle[i].right;
        if (right.length() == 1)
            if (get_type(right[0]) == 1)
                return false;
        for (int j = 0; j < right.length() - 1; j++)
```

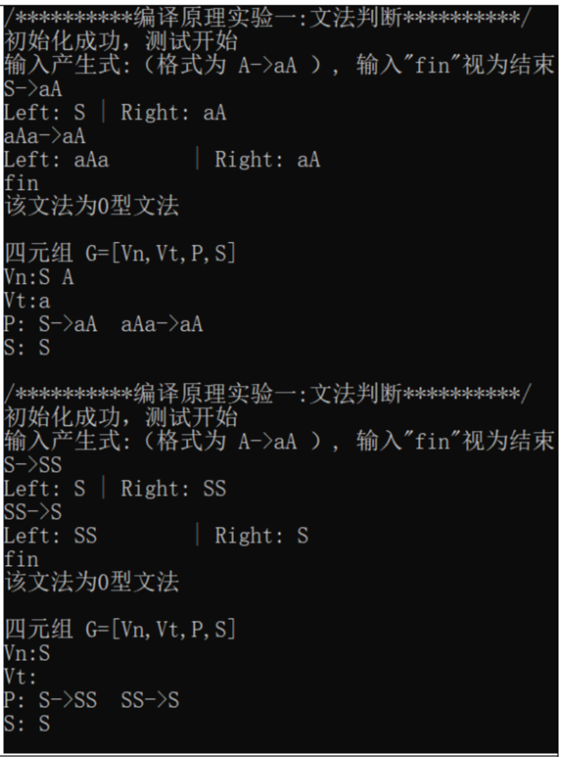
```
            if (get_type(right [ j ] ) == 1 )
                return false ;
    }
    return true ;
}

int analysis_result()
{
    if (!isZero()) return -1;
    if (!isOne()) return 0;
    if (!isTwo()) return 1;
    if (isLeftThree()) return 3;   // if neither is left linear
    if (isRightThree()) return 4;  // not right linear
    return 2;                          // It is not type 3 but type 2
}

void init()
{
    VN. clear();
    VT.clear();
    EP.clear();
    principle. clear();
    memset(type, 0, sizeof (type));
    cout << "/********** Compilation Principle Experiment 1 : Grammar Judgment **********/\
nInitialization succeeded, test started " << endl;
}
```

Experimental results:



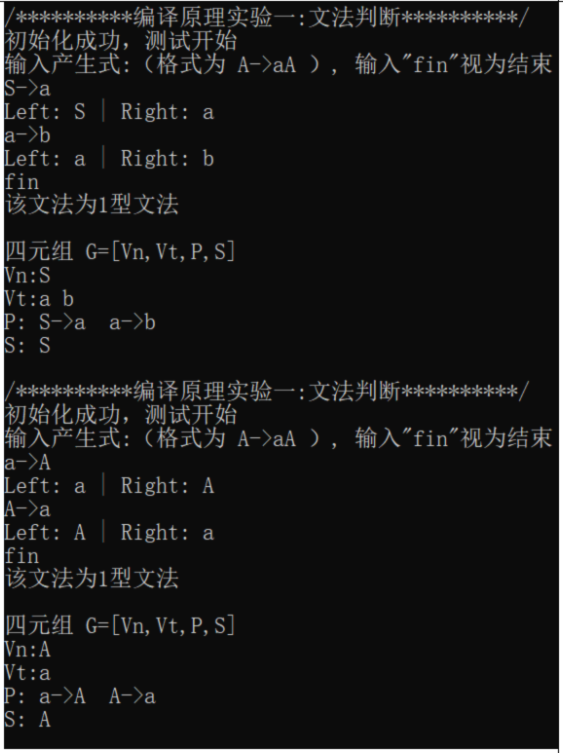
```
/**********编译原理实验一:文法判断**********/
初始化成功，测试开始
输入产生式:（格式为 A->aA ），输入"fin"视为结束
S
Left: S | Right:
fin
输入不规范，非Chomsky文法

/**********编译原理实验一:文法判断**********/
初始化成功，测试开始
输入产生式:（格式为 A->aA ），输入"fin"视为结束
S->
Left: S | Right:
fin
输入不规范，非Chomsky文法

/**********编译原理实验一:文法判断**********/
初始化成功，测试开始
输入产生式:（格式为 A->aA ），输入"fin"视为结束
->S
Left:   | Right: S
fin
输入不规范，非Chomsky文法
```

non-Chomsky grammar



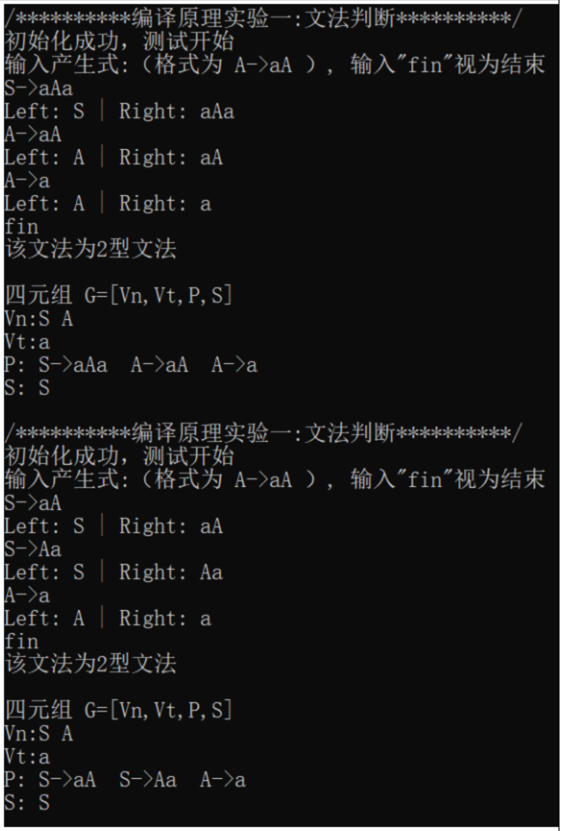
```
/**********编译原理实验一:文法判断**********/
初始化成功，测试开始
输入产生式:（格式为 A->aA ），输入"fin"视为结束
S->aA
Left: S | Right: aA
aAa->aA
Left: aAa        | Right: aA
fin
该文法为0型文法

四元组 G=[Vn,Vt,P,S]
Vn:S A
Vt:a
P: S->aA  aAa->aA
S: S

/**********编译原理实验一:文法判断**********/
初始化成功，测试开始
输入产生式:（格式为 A->aA ），输入"fin"视为结束
S->SS
Left: S | Right: SS
SS->S
Left: SS         | Right: S
fin
该文法为0型文法

四元组 G=[Vn,Vt,P,S]
Vn:S
Vt:
P: S->SS  SS->S
S: S
```

Type 0 grammar



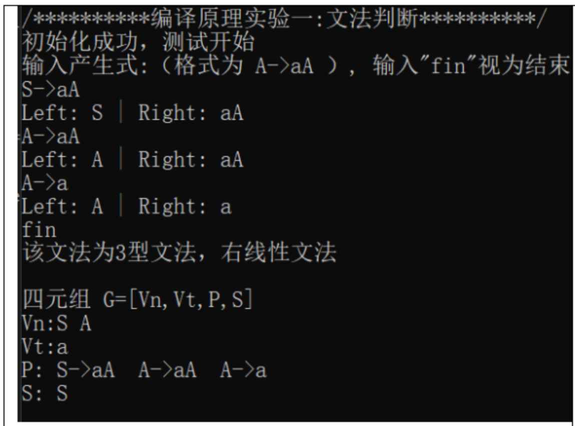
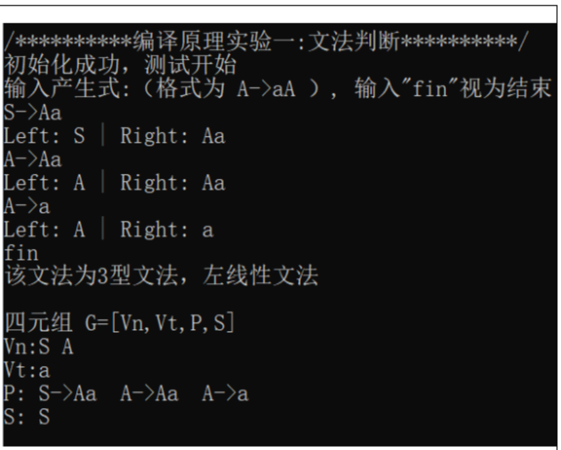
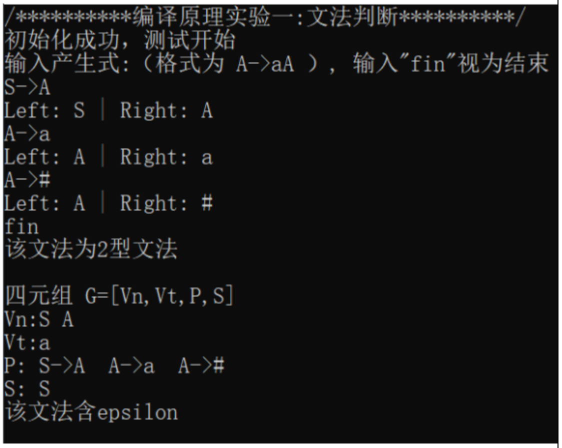
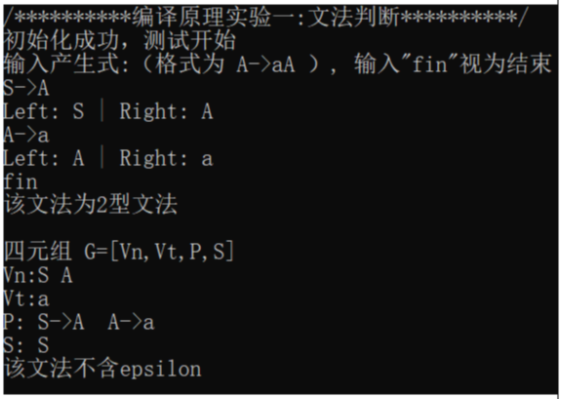
```
/**********编译原理实验一:文法判断**********/
初始化成功，测试开始
输入产生式:（格式为 A->aA ），输入"fin"视为结束
S->a
Left: S | Right: a
a->b
Left: a | Right: b
fin
该文法为1型文法

四元组 G=[Vn,Vt,P,S]
Vn:S
Vt:a b
P: S->a  a->b
S: S

/**********编译原理实验一:文法判断**********/
初始化成功，测试开始
输入产生式:（格式为 A->aA ），输入"fin"视为结束
a->A
Left: a | Right: A
A->a
Left: A | Right: a
fin
该文法为1型文法

四元组 G=[Vn,Vt,P,S]
Vn:A
Vt:a
P: a->A  A->a
S: A
```

Type 1 grammar



```
/**********编译原理实验一:文法判断**********/
初始化成功，测试开始
输入产生式:（格式为 A->aA ），输入"fin"视为结束
S->aAa
Left: S | Right: aAa
A->aA
Left: A | Right: aA
A->a
Left: A | Right: a
fin
该文法为2型文法

四元组 G=[Vn,Vt,P,S]
Vn:S A
Vt:a
P: S->aAa  A->aA  A->a
S: S

/**********编译原理实验一:文法判断**********/
初始化成功，测试开始
输入产生式:（格式为 A->aA ），输入"fin"视为结束
S->aA
Left: S | Right: aA
S->Aa
Left: S | Right: Aa
A->a
Left: A | Right: a
fin
该文法为2型文法

四元组 G=[Vn,Vt,P,S]
Vn:S A
Vt:a
P: S->aA  S->Aa  A->a
S: S
```

Type 2 grammar

 /**********编译原理实验一:文法判断**********/ 初始化成功，测试开始 输入产生式：（格式为 A->aA ），输入"fin"视为结束 S->aA Left: S | Right: aA A->aA Left: A | Right: aA A->a Left: A | Right: a fin 该文法为3型文法，右线性文法 四元组 G=[Vn,Vt,P,S] Vn:S A Vt:a P: S->aA A->aA A->a S: S	 /**********编译原理实验一:文法判断**********/ 初始化成功，测试开始 输入产生式：（格式为 A->aA ），输入"fin"视为结束 S->Aa Left: S | Right: Aa A->Aa Left: A | Right: Aa A->a Left: A | Right: a fin 该文法为3型文法，左线性文法 四元组 G=[Vn,Vt,P,S] Vn:S A Vt:a P: S->Aa A->Aa A->a S: S
Type 3 right linear grammar	Type 3 Left Linear Grammar
 /**********编译原理实验一:文法判断**********/ 初始化成功，测试开始 输入产生式：（格式为 A->aA ），输入"fin"视为结束 S->A Left: S | Right: A A->a Left: A | Right: a A-># Left: A | Right: # fin 该文法为2型文法 四元组 G=[Vn,Vt,P,S] Vn:S A Vt:a P: S->A A->a A-># S: S 该文法含epsilon	 /**********编译原理实验一:文法判断**********/ 初始化成功，测试开始 输入产生式：（格式为 A->aA ），输入"fin"视为结束 S->A Left: S | Right: A A->a Left: A | Right: a fin 该文法为2型文法 四元组 G=[Vn,Vt,P,S] Vn:S A Vt:a P: S->A A->a S: S 该文法不含epsilon
With ″ε″, the display grammar contains e psilon (# stands for empty)	Without ″ε″, the display grammar does not contain e psilon