1. 算法整体设计：

1.1 进行文法和句子的输入以及合法性的验证
1.2 将输入的上下文无关文法转化为乔姆斯基范式
1.3 以转化得到的文法的乔姆斯基范式为基础，广度优先的进行 2n-1 步最右派生（n 为所给定的句子的长度），若能够得到给定的句子则说明该句子是由该文法生成的。

2. 算法解释：

2.1 采用搜索算法的原因：上下文无关文法对应一个下推自动机，判断该句子是否由该文法生成即等价于该句子能否被该对应的自动机识别。空栈接受型的下推自动机其压栈出栈和状态转移的过程可以看作是根据文法不断派生以匹配输入句子的过程，这个过程可以用以每步派生为单位的搜索算法来模拟
2.2 将输入的文法转化为乔姆斯基范式的原因：在选择搜索算法时，我考虑了深度优先算法和广度优先算法两种方式。我最先选择的是深度优先算法，因为可以仅保留当前的派生状态不用记录其他分支，节省空间上的开销。思路为：判断是否匹配=>判断是否剪枝和回退=>进行派生。循环进行此操作直到匹配成功或者不能够再进行回退（栈空）为止，即可得出结论。
但是这会出现一个问题，深度优先可能会沿着某一条路径无限的派生下去陷入死循环，因此无法得出结论。我转而考虑了广度优先搜索，但是也存在同样的问题，可能会出现某条分支无限派生下去的情况。
经过分析，我认为这是文法的形式所导致的问题，像 S=>SS 等形式的生成式会导致循环的发生。因此联想到了规范化文法的形式，查阅资料得，乔姆斯基范式总是能够通过 2n-1 步的派生得到相应的句子，这为搜索算法的搜索深度给出了限制，保证了其不会无限的搜索下去。
2.3 采用广度优先搜索的原因：在有了深度的限定后，无论是深度优先的搜索还是广度优先的搜索都能够得出结论，最终我还是认为广度优先更方便一些，一是因为可以采用递归的形式编写，并且不用记录回退状态和分支情况，代码会更简单清晰，二是我认为这样更符合非确定型自动机状态转移的情况。

3. 算法具体设计

3.1 文法和句子的输入和保存：使用 char 来存储起始符，vector 来存储变元集和终结符集合，map<char, vector<string>>来存储生成式。
3.2 将上下文无关文法转化为乔姆斯基范式：

a. 添加开始变元以及规则
b. 消除所有的 ε 规则
c. 消除所有的 A->B 规则
d.添加变元消除 A->BCD 规则

3.3 进行广度优先搜索：设计函数 bool(string currentS, Grammer G, int i)，currentS 为当前表达式，G 为乔姆斯基范式的上下文无关文法，i 为当前搜索的深度，其思路为对 currentS 进行最右派生，生成其可能派生出的表达式，再递归调用该函数，直到不能派

生或者深度到达 2n-1 为止，其返回值为真如果其递归调用的子函数中至少有一个为真，表明句子是否能够被当前语法派生出来。

4. 代码实现

我并没有能够完整的实现上述的算法，下述代码仅仅完成到深度优先搜索的部分，没有实现将上下文无关文法转化为乔姆斯基范式，也没有在此基础上进行广度优先搜索并得出结论。这是因为我在以下几个方面遇到了困难：

①**错误输入的处理：**对于可能出现的错误输入的判断和处理比较繁琐和耗时，尤其是对于生成式的处理，不仅要判断其字符是否合法，还要判断其是否与变元集和终结符集相匹配，以及格式问题。

②**乔姆斯基范式的转化：**在转化的过程中涉及到新变元的添加，新产生式的添加和产生式的修改，而变元由输入决定，因此新变元的确定要排除输入。另外由于变元和终结符的区分依赖于大小写，字母数量的限制问题也让我有点困惑。乔姆斯基范式转化的整体过程也较为复杂，耗费心力，我没有能够完成。

```
#include <iostream>
#include <vector>
#include <conio.h>
#include <map>
#include <string.h>
#include <stack>
using namespace std;

int main() {
    /*进行文法的输入*/
    char initChar;
    vector<char> variaSet;
    vector<char> endSet;
    map<char, vector<string>>
generatives;
    //输入起始符
    cout << "请输入起始符：";
    cin >> initChar;
    variaSet.push_back(initChar);
    //输入中间变元集
    while (true) {
        char isEnd;
        cout << "是否继续输入中间变元?
(y/n)：";
        while (true) {
            isEnd = _getch();
            if (isEnd == 'y' || isEnd ==
'n') {
                break;
            }
        }
        if (isEnd == 'n') {
            break;
        }
        char varia;
        cout << endl << "请输入中间变
元：";
        cin >> varia;
        variaSet.push_back(varia);
    }
    cout << endl;
    //输入终结符集
    while (true) {
        char isEnd;
        cout << "是否继续输入终结符?
(y/n)：";
        while (true) {
            isEnd = _getch();
            if (isEnd == 'y' || isEnd ==
'n') {
                break;
            }
        }
        if (isEnd == 'n') {
```

```
            break;
        }
        char end;
        cout << endl << "请输入终结符：
";
        cin >> end;
        endSet.push_back(end);
    }
    cout << endl;
    //输入生成式
    while (true) {
        char isEnd;
        cout << "是否继续输入生成式？
(y/n)：";
        while (true) {
            isEnd = _getch();
            if (isEnd == 'y' || isEnd ==
'n') {
                break;
            }
        }
        if (isEnd == 'n') {
            break;
        }
        char leftSide;
        string rightSide;
        cout << endl << "请输入生成式左
端字符：";
        cin >> leftSide;
        cout << "请输入生成式右端字符串
（$字符代表空串）：";
        cin >> rightSide;
        if (generatives.count(leftSide)
== 0) {
            vector<string> temp;
            generatives[leftSide] =
temp;
        }

    generatives[leftSide].push_back(right
Side);
    }

    /*打印输入的上下文无关文法*/
    //打印起始符
    cout << endl << "起始符：" <<
initChar << endl;
    //打印变元集
    cout << "变元集：";
    for (int i = 0; i < variaSet.size();
i++) {
        cout << variaSet[i] << " ";
    }
    cout << endl;
    //打印终结符集
    cout << "终结符集：";
    for (int i = 0; i < endSet.size();
i++) {
        cout << endSet[i] << " ";
    }
    cout << endl;
    //打印生成式集
    cout << "生成式：" << endl;
    for (int i = 0; i < variaSet.size();
i++) {
        cout << variaSet[i] << " -> ";
        for (int j = 0; j <
generatives[variaSet[i]].size(); j++) {
            cout <<
generatives[variaSet[i]][j];
            if (j !=
generatives[variaSet[i]].size() - 1) {
                cout << "/";
            }
        }
        cout << endl;
    }

    /*进行上下文无关语言的识别*/
    //输入需要判别的句子
    string sentence;
    cout << "请输入要判别的句子：";
    cin >> sentence;
    //初始化栈、轨迹和当前可能的分支
    vector<char> S;
    stack<string> trace;
    stack<int> branches;
    S.push_back(initChar);
```

```
    trace.push(string("" + initChar));
    branches.push(generatives[initChar].size());
    //进行深度优先搜索，寻找匹配结果
    while (true) {
        //判断当前表达式是否匹配
        bool isMatch = true;
        if (sentence.size() == S.size()) {
            for (int i = 0; i < S.size(); i++) {
                if (S[i] != sentence[i]) {
                    isMatch = false;
                }
            }
        }
        else {
            isMatch = false;
        }
        //若匹配，则退出，该语句是该文法产生的
        if (isMatch) {
            cout << "该语句是由该上下文无关文法所产生的" << endl;
            break;
        }
        //若不匹配，则判断是否可能沿着当前路径派生出该句子
        bool canGener = true;

        if (!canGener) {
            trace.pop();
            if (trace.empty()) {
                cout << "该语句不能由该上下文无关文法生成" << endl;
                break;//栈已经弹空，说明不能再回退，该语句不能由该上下文无关文法生成
            }
            else {
                trace.pop();
                S.clear();
                branches.pop();
                for (int i = 0; i < trace.top().size(); i++) {

                    S.push_back(trace.top()[i]);
                }
            }
        }//若不可能匹配，则进行回退
        else {
            vector<char> temS;
            int recLastVaria = 0;
            for (int i = 0; i < S.size(); i++) {
                temS.push_back(S[i]);
                if (S[i] >= 'A' && S[i] <= 'Z') {
                    recLastVaria = i;
                }
            }
            S.clear();
            for (int i = 0; i < recLastVaria; i++) {
                S.push_back(temS[i]);
            }
        }//若可能匹配，进行最右派生
    }
}
```