实验二. 自上而下语法分析

设计思想

根据对自上而下语法分析的理论知识的学习，可以知道自上而下语法分析的两种实现方法：递归下降子程序法以及预测分析程序法，本实验采用后者预测分析法。

本实验对PL0文法的表达式文法进行设计自上而下语法分析，表达式巴斯克范式如下：

$$
\begin{aligned}
&< \text{表达式} >::= [+|-] < \text{项} > \{< \text{加法运算符} >< \text{项} >\}\\
&< \text{项} >::=< \text{因子} > \{< \text{乘法运算符} >< \text{因子} >\}\\
&< \text{因子} >::=< \text{标识符} > | < \text{无符号整数} > |'('< \text{表达式} >')'\\
&< \text{加法运算符} >::= +|-\\
&< \text{乘法运算符} >::= *|/
\end{aligned}
$$

对以上文字描述的文法可以给出对应的英文表示，以及对应的非终结符的 $First$集合 和 $Follow$集合 ：

$$
\begin{aligned}
&< Expression >::= [+|-] < Term > \{< Addop >< Term >\}\\
&< Term >::=< Factor > \{< Mulop >< Factor >\}\\
&< Factor >::=< Ident > | < UnsignInt > |(< Expression >)\\
&< Addop >::= +|-\\
&< Mulop >::= *|/
\end{aligned}
$$

该文法的非终结符结合以及终结符集合如下：

$$
\begin{aligned}
V_T &= \{+, -, *, /, (,), Ident, UnsignInt\}\\
V_N &= \{Expression, Term, Addop, Factor, Mulop\}
\end{aligned}
$$

$First$集合 ：

$$
\begin{aligned}
&First(Expression) = \{+, -, Ident, UnsignInt, (\}\\
&First(Term) = \{Ident, UnsignInt, (\}\\
&First(Factor) = \{Ident, UnsignInt, (\}\\
&First(Addop) = \{+, -\}\\
&First(Mulop) = \{*, /\}
\end{aligned}
$$

$Follow$集合 ：

$$
\begin{aligned}
&Follow(Expression) = \{\#,)\}\\
&Follow(Term) = \{\#, +, -,)\}\\
&Follow(Factor) = \{\#, +, -, *, /,)\}\\
&Follow(Addop) = \{Ident, Unsigned, (\}\\
&Follow(Mulop) = \{Ident, UnsignInt, (\}
\end{aligned}
$$

使用巴斯克范式表示的文法来设计递归下降子程序分析方法很方便，但是，对于设计预测分析方法的程序，显然要转化为符合规范的LL1文法，此时定义终结符和非终结符集合如下：

$$
\begin{aligned}
V_T &= \{+, -, *, /, (,), i, u\}\\
V_N &= \{E, E', T, T', F\}
\end{aligned}
$$

处理后得到的新文法如下：（显然是LL1文法）

$$
\begin{aligned}
&E \to TE'|+TE'|-TE'\\
&E' \to +TE'|-TE'|\epsilon\\
&T \to FT'\\
&T' \to *FT'|/FT'|\epsilon\\
&F \to i|u|(E)
\end{aligned}
$$

此时可以求出非终结符的 $First$集合 以及 $Follow$集合：

$$
\begin{aligned}
&Frist(E) = \{+, -, i, u, (\} \\
&Frist(E') = \{+, -, \epsilon\} \\
&Frist(T) = \{i, u, (\} \\
&Frist(T') = \{*, /, \epsilon\} \\
&Frist(F) = \{i, u, (\}
\end{aligned}
$$

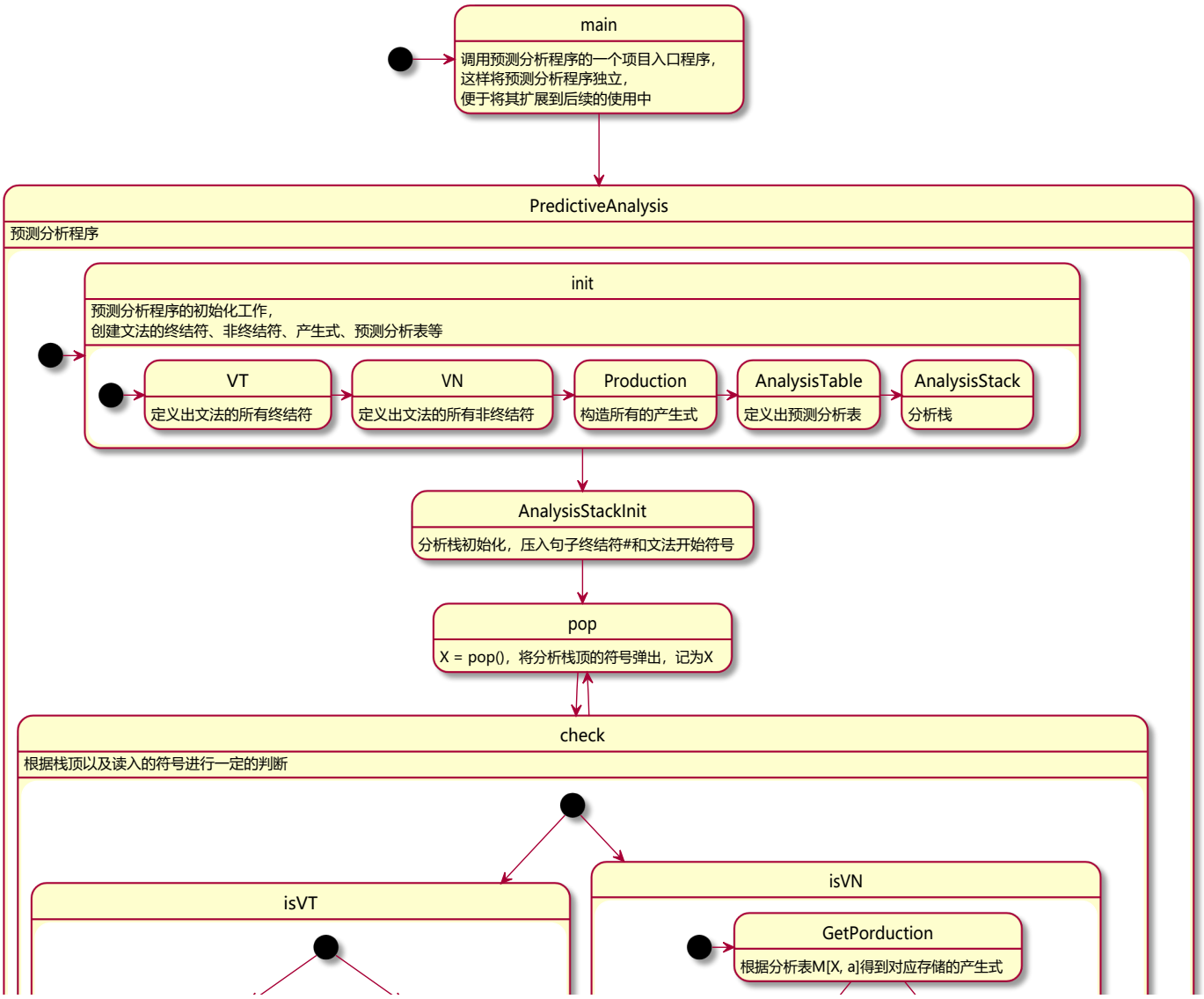
$$
\begin{aligned}
&Follow(E) = \{\#,)\} \\
&Follow(E') = \{\#,)\} \\
&Follow(T) = \{\#,), +, -\} \\
&Follow(T') = \{\#,), +, -\} \\
&Follow(F) = \{\#,), +, -, *, /\}
\end{aligned}
$$

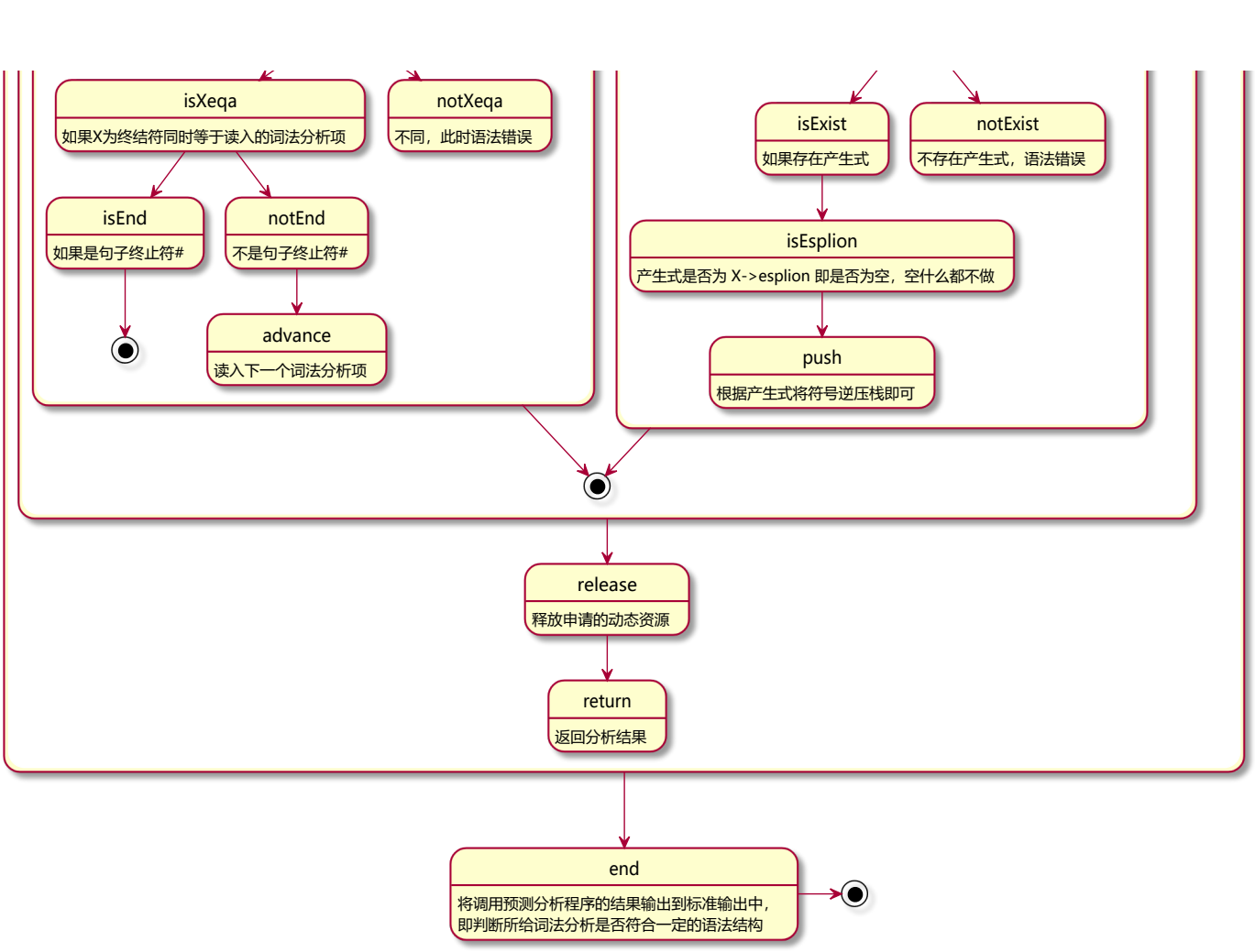
对于预测分析程序，最重要的组成就是 **预测分析表** $M[A, a]$，根据预测分析表的构造算法可以得到上述文法的预测分析表 $M[A, a]$：

分析表	+	-	*	/	(	)	i	u	#
E	$E \to +TE'$	$E \to -TE'$			$E \to TE'$		$E \to TE'$	$E \to TE'$	
E'	$E' \to +TE'$	$E' \to -TE'$				$E' \to \epsilon$			$E' \to \epsilon$
T					$T \to FT'$		$T \to FT'$	$T \to FT'$	
T'	$T' \to \epsilon$	$T' \to \epsilon$	$T' \to *FT'$	$T' \to /FT'$		$T' \to \epsilon$			$T' \to \epsilon$
F					$F \to (E)$		$F \to i$	$F \to u$	

得到预测分析表后，按照预测分析程序中总控程序的框架进行编写代码即可完成预测分析程序的实现。

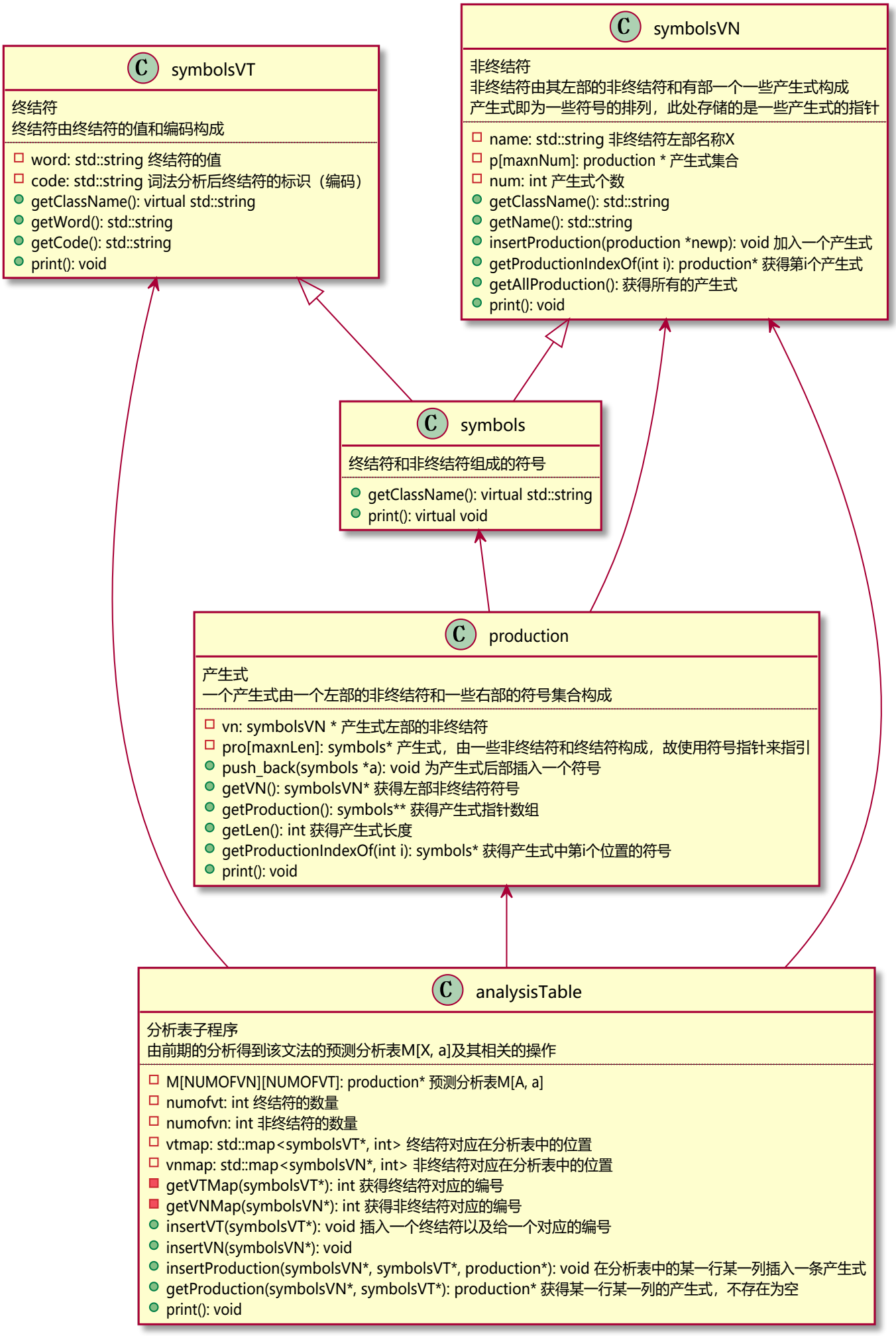
算法流程





源程序

整个预测分析程序所设计到的类以及相互的关系如下：


symbolsVT
终结符
终结符由终结符的值和编码构成
word: std::string 终结符的值
code: std::string 词法分析后终结符的标识（编码）
getClassName(): virtual std::string
getWord(): std::string
getCode(): std::string
print(): void
symbolsVN
非终结符
非终结符由其左部的非终结符和有部一个一些产生式构成
产生式即为一些符号的排列，此处存储的是一些产生式的指针
name: std::string 非终结符左部名称X
p[maxnNum]: production * 产生式集合
num: int 产生式个数
getClassName(): std::string
getName(): std::string
insertProduction(production *newp): void 加入一个产生式
getProductionIndexOf(int i): production* 获得第i个产生式
getAllProduction(): 获得所有的产生式
print(): void
symbols
终结符和非终结符组成的符号
getClassName(): virtual std::string
print(): virtual void
production
产生式
一个产生式由一个左部的非终结符和一些右部的符号集合构成
vn: symbolsVN * 产生式左部的非终结符
pro[maxnLen]: symbols* 产生式，由一些非终结符和终结符构成，故使用符号指针来指引
push_back(symbols *a): void 为产生式后部插入一个符号
getVN(): symbolsVN* 获得左部非终结符符号
getProduction(): symbols** 获得产生式指针数组
getLen(): int 获得产生式长度
getProductionIndexOf(int i): symbols* 获得产生式中第i个位置的符号
print(): void
analysisTable
分析表子程序
由前期的分析得到该文法的预测分析表M[X, a]及其相关的操作
M[NUMOFVN][NUMOFVT]: production* 预测分析表M[A, a]
numofvt: int 终结符的数量
numofvn: int 非终结符的数量
vtmap: std::map<symbolsVT*, int> 终结符对应在分析表中的位置
vnmap: std::map<symbolsVN*, int> 非终结符对应在分析表中的位置
getVTMap(symbolsVT*): int 获得终结符对应的编号
getVNMap(symbolsVN*): int 获得非终结符对应的编号
insertVT(symbolsVT*): void 插入一个终结符以及给一个对应的编号
insertVN(symbolsVN*): void
insertProduction(symbolsVN*, symbolsVT*, production*): void 在分析表中的某一行某一列插入一条产生式
getProduction(symbolsVN*, symbolsVT*): production* 获得某一行某一列的产生式，不存在为空
print(): void

项目地址

基础符号类

基础符号类作为一个符号类的基类，主要用途为使用基类指针来指引子类的终结符或非终结符，简化后续的操作。

```
// symbols.h
#ifndef symbols_h
#define symbols_h
#include<iostream>
/*
终结符和非终结符的一个共同的基类
这样可以通过基类来引用终结符和非终结符

*/
class symbols
{
private:
    /* data */
public:
    symbols(){};
    virtual ~symbols(){};
    virtual std::string getClassName(){
        return "symbols";
    }
    virtual void print(){};
};

#endif /*symbols_h*/
```

终结符类

终结符类继承至基础符号类，终结符由终结符的值和编码构成。

```
// symbolsVT.h
#ifndef symbolsVT_h
#define symbolsVT_h
#include<iostream>
#include"symbols.h"
/*
一个终结符
终结符由终结符的值和编码构成

*/
class symbolsVT : public symbols{
private:
    std::string word;   //终结符的值
    std::string code;   //词法分析后终结符的标识（编码）
public:
    symbolsVT(){}
    symbolsVT(std::string W, std::string C):word(W), code(C) {
        std::cerr<< "V_T: " << word << " " << code << " created..." << std::endl;
    }
    ~symbolsVT() {}
    std::string getClassName();
    std::string getWord();
    std::string getCode();
    void print();
};

#endif /*symbolsVT_h*/
```

```cpp
// symbolsVT.cpp
#include<iostream>
#include"symbolsVT.h"

std::string symbolsVT::getClassName(){
    return "VT";
}
std::string symbolsVT::getWord(){
    return word;
}
std::string symbolsVT::getCode(){
    return code;
}
void symbolsVT::print(){
    std::cerr << "VT: " << word << " " << code << std::endl;
}
```

非终结符类

非终结符类与终结符类一样继承至基础符号类，非终结符由其左部的非终结符和有部一个一些产生式构成，产生式即为一些符号的排列，此处存储的是一些产生式的指针。

```cpp
// symbolsVN.h
#ifndef symbolsVN_h
#define symbolsVN_h
#include<iostream>
#include"symbols.h"
#include"production.h"
const int maxnNum = 5;     // 一个终结符所有的产生式的数量
class production;
/*
非终结符
非终结符由其左部的非终结符和有部一个一些产生式构成
产生式即为一些符号的排列，此处存储的是一些产生式的指针
*/

class symbolsVN: public symbols{
private:
    std::string name;                                           // 非终结符左部名称X
    production *p[maxnNum];                                     // 产生式集合
    int num;
public:
    symbolsVN();
    symbolsVN(std::string);
    ~symbolsVN() {}
    std::string getClassName();
    std::string getName();
    void insertProduction(production *newp);    // 加入一个产生式
    production* getProductionIndexOf(int i);    // 获得第i个产生式
    production** getAllProduction();                    // 获得所有的产生式
    void print();
};

#endif /*symbolsVN_h*/
```

```cpp
// symbolsVN.cpp
#include<iostream>
#include"symbolsVN.h"

symbolsVN::symbolsVN(){
    num = 0;
}
symbolsVN::symbolsVN(std::string n):name(n){
    symbolsVN();
    std::cerr << "V_N: " << name << " created..." << std::endl;
}

std::string symbolsVN::getClassName(){
    return "VN";
}
std::string symbolsVN::getName(){
    return name;
}
void symbolsVN::insertProduction(production *newp){
    p[num++] = newp;
    return;
}
production* symbolsVN::getProductionIndexOf(int i){
    if(i >= num){
        std::cerr << "index overflow..." << std::endl;
        return NULL;
    }
    return p[i];
}
production** symbolsVN::getAllProduction(){
    return p;
}
void symbolsVN::print(){
    std::cerr << "VN: " << name << std::endl;
    std::cerr << "ALL production: " << std::endl;
    for(int i = 0; i < num; ++i){
        std::cerr << name << " \\to ";
        p[i]->print();
    }
    std::cerr << std::endl;
}
```

产生式类

一个产生式由一个左部的非终结符和一些右部的符号集合构成。

```h
// production.h
#ifndef production_h
#define production_h
#include<iostream>
#include"symbols.h"
#include"symbolsVT.h"
#include"symbolsVN.h"
/*
产生式
一个产生式由一个左部的非终结符和一些右部的符号集合构成
*/

const int maxnLen = 10;        // 一个产生式的右部符号的数量
class symbolsVN;
class production
{
private:
    symbolsVN *vn;                                  // 产生式左部的非终结符
    symbols *pro[maxnLen];                      // 产生式，由一些非终结符和终结符构成，故使用符号指针来指引
    int len;
public:
    production();
    production(symbolsVN *v);
    ~production(){}
    void push_back(symbols *a);         // 为产生式后部插入一个符号
    symbolsVN* getVN();                         // 获得左部非终结符符号
    symbols** getProduction();          // 获得产生式指针数组
    int getLen();                                   // 获得产生式长度
    symbols* getProductionIndexOf(int i); // 获得产生式中第i个位置的符号
    void print();
};

#endif /*production_h*/
```

```cpp
// production.cpp
#include<iostream>
#include"production.h"
#include"symbolsVT.h"
#include"symbolsVN.h"

production::production(){
    len = 0;
}
production::production(symbolsVN *v){
    vn = v;
    production();
    std::cerr << "A production of " << vn->getName() << " has created..." << std::endl;
}
void production::push_back(symbols *a){
    pro[len++] = a;
}
symbolsVN* production::getVN(){
    return vn;
}
symbols** production::getProduction(){
    return pro;
}
symbols* production::getProductionIndexOf(int i){
    if(i >= len){
        std::cerr << "index Overflow..." << std::endl;
        return NULL;
    }
    return pro[i];
}
int production::getLen(){
    return len;
}
void production::print(){
    std::cerr << vn->getName() << "->";
    for(int i = 0; i < len; ++i){
        if(pro[i]->getClassName() == "VT"){
            std::cerr << ((symbolsVT*)pro[i])->getWord();
        }
        else{
            std::cerr << ((symbolsVN*)pro[i])->getName();
        }
    }
    // std::cerr << std::endl;
}
```

分析表类

由前期的分析得到该文法的预测分析表M[X, a]及其相关的操作。

```h
// analysisTable.h
#ifndef analysisTable_h
#define analysisTable_h
#include<map>
#include"symbols.h"
#include"symbolsVN.h"
#include"symbolsVT.h"
#include"production.h"
const int NUMOFVT = 10;
const int NUMOFVN = 10;
/*
分析表子程序
由前期的分析得到该文法的预测分析表M[X, a]及其相关的操作

*/

class analysisTable
{
private:
    production* M[NUMOFVN][NUMOFVT];    // 预测分析表M[A, a]
    int numofvt;                        // 终结符的数量
    int numofvn;                        // 非终结符的数量

    std::map<symbolsVT*, int> vtmap;    // 终结符对应在分析表中的位置
    std::map<symbolsVN*, int> vnmap;    // 非终结符对应在分析表中的位置

    int getVTMap(symbolsVT*);                   // 获得终结符对应的编号
    int getVNMap(symbolsVN*);                   // 获得非终结符对应的编号
public:
    analysisTable();
    analysisTable(int, int);
    ~analysisTable() {}
    void insertVT(symbolsVT*);                  // 插入一个终结符以及给一个对应的编号
    void insertVN(symbolsVN*);                  // 插入一个非终结符以及给一个对应的编号
    void insertProduction(symbolsVN*, symbolsVT*, production*); // 在分析表中的某一行某一列插入一条产生式
    production* getProduction(symbolsVN*, symbolsVT*);  // 获得某一行某一列的产生式，不存在为空
    void print();
};

#endif /*analysisTable_h*/
```

```cpp
// analysisTable.cpp
#include<iostream>
#include<string.h>
#include"analysisTable.h"

analysisTable::analysisTable(){
    memset(M, 0, sizeof M);
    numofvn = numofvt = 0;
    vtmap.clear();
    vnmap.clear();
}
analysisTable::analysisTable(int nvt, int nvn):numofvt(nvt), numofvn(nvn){
    analysisTable();
    // for(int i = 0; i < numofvn; ++i){
    //     for(int j = 0; j < numofvt; ++j){
    //         M[i][j] = nullptr;
    //     }
    // }
}
void analysisTable::insertVT(symbolsVT* vt){
    vtmap[vt] = numofvt++;
}
void analysisTable::insertVN(symbolsVN* vn){
    vnmap[vn] = numofvn++;
}
int analysisTable::getVTMap(symbolsVT* vt){
    return vtmap[vt];
}
int analysisTable::getVNMap(symbolsVN* vn){
    return vnmap[vn];
}
void analysisTable::insertProduction(symbolsVN* vn, symbolsVT* vt, production* p){
    M[getVNMap(vn)][getVTMap(vt)] = p;
}
production* analysisTable::getProduction(symbolsVN* X, symbolsVT* a){
    return M[getVNMap(X)][getVTMap(a)];
}

void analysisTable::print(){
    std::cerr << numofvn << " " << numofvt << std::endl;
    for(int i = 0; i < numofvn; ++i){
        for(int j = 0; j < numofvt; ++j){
            std::cerr << M[i][j] << "\t";
        }
        std::cerr << std::endl;
    }
    std::cerr << std::endl;
    for(int i = 0; i < numofvn; ++i){
        for(int j = 0; j < numofvt; ++j){
            if(M[i][j] != NULL){
                M[i][j]->print();
            }
            std::cerr << "\t";
        }
        std::cerr << std::endl;
    }
}
```

预测分析法总控程序

预测分析法的核心部分，初始化由前期的分析得到，确定具体的符号集、预测分析表等等，然后根据总控程序的结构实现即可，最后调用即可。

```cpp
// predictiveAnalysis.cpp
#include<iostream>
#include<cstdio>
#include<string.h>
#include"symbols.h"
#include"symbolsVN.cpp"
#include"symbolsVT.cpp"
#include"production.cpp"
#include"analysisTable.cpp"
const int maxnAnalysisStack = 1e2 + 5;

// 定义出文法的所有终结符
symbolsVT* PLUS = new symbolsVT("+", "plus");
symbolsVT* MINUS = new symbolsVT("-", "minus");
symbolsVT* times = new symbolsVT("*", "times");
symbolsVT* slash = new symbolsVT("/", "slash");
symbolsVT* lparen = new symbolsVT("(", "lapren");
symbolsVT* rparen = new symbolsVT(")", "rparen");
symbolsVT* ident = new symbolsVT("i", "ident");
symbolsVT* unsignint = new symbolsVT("u", "unsignint");
symbolsVT* END = new symbolsVT("#", "end");
symbolsVT* epslion = new symbolsVT("e", "epslion");
// 定义出文法的所有非终结符
symbolsVN* E = new symbolsVN("E");
symbolsVN* Edot = new symbolsVN("E'");
symbolsVN* T = new symbolsVN("T");
symbolsVN* Tdot = new symbolsVN("T'");
symbolsVN* F = new symbolsVN("F");

// 构造所有的产生式
production* Eporduction[3];
production* Edotproduction[3];
production* Tproduction[1];
production* Tdotproduction[3];
production* Fproduction[3];

// 定义出预测分析表
analysisTable AnalysisTable;

// 分析栈
symbols* analysisStack[maxnAnalysisStack];
int top;
void init(){
    // 初始化所有变量
        // 根据文法的不同，得到的分析表的结构也不同，此时初始化部分也不同

    // 定义出预测分析表
        // 为预测分析表插入终结符、非终结符
    AnalysisTable.insertVT(PLUS);
    AnalysisTable.insertVT(MINUS);
    AnalysisTable.insertVT(times);
    AnalysisTable.insertVT(slash);
    AnalysisTable.insertVT(lparen);
    AnalysisTable.insertVT(rparen);
    AnalysisTable.insertVT(ident);
    AnalysisTable.insertVT(unsignint);
    AnalysisTable.insertVT(END);

    AnalysisTable.insertVN(E);
    AnalysisTable.insertVN(Edot);
    AnalysisTable.insertVN(T);
    AnalysisTable.insertVN(Tdot);
    AnalysisTable.insertVN(F);

    // 根据文法定义E的三条产生式，同理处理其他的产生式
    for(int i = 0; i < 3; ++i)Eporduction[i] = new production(E);

    Eporduction[0]->push_back(T);
    Eporduction[0]->push_back(Edot);
    Eporduction[1]->push_back(PLUS);
    Eporduction[1]->push_back(T);
    Eporduction[1]->push_back(Edot);
    Eporduction[2]->push_back(MINUS);
    Eporduction[2]->push_back(T);
    Eporduction[2]->push_back(Edot);
    Eporduction[0]->print(); Eporduction[1]->print(); Eporduction[2]->print();
    E->insertProduction(Eporduction[0]);
    E->insertProduction(Eporduction[1]);
    E->insertProduction(Eporduction[2]);
```

```
    for(int i = 0; i < 3; ++i)Edotproduction[i] = new production(Edot);
    Edotproduction[0]->push_back(PLUS);
    Edotproduction[0]->push_back(T);
    Edotproduction[0]->push_back(Edot);
    Edotproduction[1]->push_back(MINUS);
    Edotproduction[1]->push_back(T);
    Edotproduction[1]->push_back(Edot);
    Edotproduction[2]->push_back(epslion);


    for(int i = 0; i < 1; ++i)Tproduction[i] = new production(T);
    Tproduction[0]->push_back(F);
    Tproduction[0]->push_back(Tdot);


    for(int i = 0; i < 3; ++i)Tdotproduction[i] = new production(Tdot);
    Tdotproduction[0]->push_back(times);
    Tdotproduction[0]->push_back(F);
    Tdotproduction[0]->push_back(Tdot);
    Tdotproduction[1]->push_back(slash);
    Tdotproduction[1]->push_back(F);
    Tdotproduction[1]->push_back(Tdot);
    Tdotproduction[2]->push_back(epslion);


    for(int i = 0; i < 3; ++i)Fproduction[i] = new production(F);
    Fproduction[0]->push_back(ident);
    Fproduction[1]->push_back(unsignint);
    Fproduction[2]->push_back(lparen);
    Fproduction[2]->push_back(E);
    Fproduction[2]->push_back(rparen);

    // 将产生式放入分析表中
    AnalysisTable.insertProduction(E, PLUS, Eporduction[1]);
    AnalysisTable.insertProduction(E, MINUS, Eporduction[2]);
    AnalysisTable.insertProduction(E, lparen, Eporduction[0]);
    AnalysisTable.insertProduction(E, ident, Eporduction[0]);
    AnalysisTable.insertProduction(E, unsignint, Eporduction[0]);

    AnalysisTable.insertProduction(Edot, PLUS, Edotproduction[0]);
    AnalysisTable.insertProduction(Edot, MINUS, Edotproduction[1]);
    AnalysisTable.insertProduction(Edot, rparen, Edotproduction[2]);
    AnalysisTable.insertProduction(Edot, END, Edotproduction[2]);

    AnalysisTable.insertProduction(T, lparen, Tproduction[0]);
    AnalysisTable.insertProduction(T, ident, Tproduction[0]);
    AnalysisTable.insertProduction(T, unsignint, Tproduction[0]);

    AnalysisTable.insertProduction(Tdot, PLUS, Tdotproduction[2]);
    AnalysisTable.insertProduction(Tdot, MINUS, Tdotproduction[2]);
    AnalysisTable.insertProduction(Tdot, times, Tdotproduction[0]);
    AnalysisTable.insertProduction(Tdot, slash, Tdotproduction[1]);
    AnalysisTable.insertProduction(Tdot, rparen, Tdotproduction[2]);
    AnalysisTable.insertProduction(Tdot, END, Tdotproduction[2]);

    AnalysisTable.insertProduction(F, lparen, Fproduction[2]);
    AnalysisTable.insertProduction(F, ident, Fproduction[0]);
    AnalysisTable.insertProduction(F, unsignint, Fproduction[1]);

    AnalysisTable.print();

        // 初始化分析栈
    memset(analysisStack, 0, sizeof analysisStack);
    top = -1;
}
void release(){
        // 释放所有的动态申请的资源
    delete PLUS;
        delete MINUS;
        delete times;
        delete slash;
        delete lparen;
        delete rparen;
        delete ident;
        delete unsignint;
        delete END;
        delete epslion;
```

```
    delete E;
        delete Edot;
        delete T;
        delete Tdot;
        delete F;
    for(int i = 0; i < 3; ++i)delete Eporduction[i];
    for(int i = 0; i < 3; ++i)delete Edotproduction[i];
    for(int i = 0; i < 1; ++i)delete Tproduction[i];
    for(int i = 0; i < 3; ++i)delete Tdotproduction[i];
    for(int i = 0; i < 3; ++i)delete Fproduction[i];
}
std::string word, code;
// char word[10], code[10];
char ch;
symbolsVT* a;
void ADVANCE(){
        // 读入一个词法分析的结果项，同时给出对应的终结符a
    // if(scanf("(%s,%s)", code, word) != -1){
    std::cin >> ch;
    if(!std::cin.eof()){
    // if(scanf("%c", &ch) != -1){
        std::getline(std::cin, code, ',');
        std::getline(std::cin, word);
        word.resize(word.size() - 1);
        // std::cin >> ch;
        std::cerr << word << " " << code << std::endl;
        if(code == "plus")a = PLUS;
        else if(code == "minus") a = MINUS;
        else if(code == "times") a = times;
        else if(code == "slash") a = slash;
        else if(code == "lparen") a = lparen;
        else if(code == "rparen") a = rparen;
        else if(code == "ident") a = ident;
        else if(code == "number") a = unsignint;
    }
    else{
        a = END;
    // if(std::cin.eof() == EOF){
        std::cerr << "hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh" << std::endl;
    }
    std::cerr << word << "_____________" << code << std::endl;
}
bool predictiveAnalysis(){
        // 预测分析程序的总控程序
    init();
    bool grammer = true;                // 表示句子是否符合一定的文法
    bool flag = true;                   // 总控程序的运行标志
    analysisStack[++top] = END; // 先将句末符加入到#分析栈中
    analysisStack[++top] = E;   // 将文法开始符加入到分析栈中
    symbols* X;         // 定义一个公共变量：符号的指针，这样可以方便的取出栈中的符号，
            //而不用管是终结符还是非终结符
    production *p;      // 定义一个产生式的指针
    ADVANCE();          // 读入一个词法分析的结果项
    while(flag){
        // std::cerr << std::endl << std::endl;
        // std::cerr << "stack: " << std::endl;;
        // for(int i = 0; i <= top; ++i){
        //     std::cerr << "> ";
        //     analysisStack[i]->print();
        // }
        X = analysisStack[top--];       // 得到分析栈的栈顶元素，pop操作
        // std::cerr << X->getClassName() << "----" << top << std::endl;
        if(X->getClassName() == "VT"){  // 如果是终结符
            if(((symbolsVT*)X)->getWord() == a->getWord()){
                // 如果分析栈顶的终结符和读入的终结符一样
                if(((symbolsVT*)X)->getWord() == END->getWord()){
                    // 如果是句末符，此时终止语法分析，语法正确
                    flag = false;
                }
                else{                                   // 读入下一个词法分析项
                    ADVANCE();
                    continue;
                }
            }
            else{               // 是终结符，但不同，显然语法错误
                grammer = false;
                flag = false;
            }
        }
```

```
        else{                   // 分析栈顶是非终结符
            // std::cerr << ((symbolsVN*)X)->getName() << " " << a->getWord() << std::endl;
            // 根据分析表的M[X, a]得到存储的产生式，存在即可以规约（替换）
            p = AnalysisTable.getProduction((symbolsVN*)X, a);
            // p->print(); std::cerr << std::endl;
            if(p != NULL){              // 非空，即为存在产生式，按照产生式替换即可
                int len = p->getLen();
                if(p->getProductionIndexOf(0)->getClassName() == "VT"){
                    // 特判 X -> epslion 的产生式，此时什么都不做
                    if(((symbolsVT*)p->getProductionIndexOf(0))->getCode() == "epslion"){
                        continue;
                    }
                }
                for(int i = len - 1; i >= 0; --i){                      // 将产生式逆序压栈即可
                    analysisStack[++top] = p->getProductionIndexOf(i);
                }
            }
            else{                               // 到达分析表不存在的位置，语法错误
                grammer = false;
                flag = false;
            }
        }
    }

    release();          // 释放资源
    return grammer;     // 返回结果，true表示句子符合一定的语法
}
```

调用

预测分析程序的调用的入口程序，根据返回值进行标准输出内容即可。

```
// main.cpp
#include<bits/stdc++.h>
#include"predictiveAnalysis.cpp"
using namespace std;
typedef long long ll;
const int mod = 1e9 + 7;
const int maxn = 1e2 + 5;
const int inf = 0x3f3f3f3f;


int main(){
    freopen("in.in", "r", stdin);
    // freopen("out.out", "w", stdout);

    if(predictiveAnalysis()) cout << "Yes,it is correct." << endl;
    else cout << "No,it is wrong." << endl;

    return 0;
}
```

调试数据

给出两组测试数据：

```
// in.in
(lparen,()
(ident,a)
(plus,+)
(number,15)
(rparen,))
(times,*)
```

```
// out.out
No,it is wrong.
```

```
// in.in
(ident,akldjsfkl)
(plus,+)
(lparen,()
(ident,alk)
(times,*)
(ident,askd)
(minus,-)
(ident,jfdkj)
(times,*)
(ident,ksfj)
(slash,/)
(ident,jsadlk)
(plus,+)
(lparen,()
(ident,a)
(slash,/)
(ident,v)
(minus,-)
(ident,d)
(plus,+)
(ident,b)
(rparen,))
(rparen,))
(slash,/)
(ident,jfj)
```

```
// out.out
Yes,it is correct.
```

实验体会

本次实验是对 **自上而下语法分析** 的一个程序实现，自上而下语法分析主要有两种方式实现： **递归下降子程序** 以及 **预测分析法** ，这两种实现的方式各有优劣，前者易于实现并且易于理解，但是，当文法很多且要从一个已有的递归下降子程序进行扩展时，需要添加的内容很多，需要添加很多的子程序；后者虽然需要前期进行大量时间计算出预测分析表，但是这样写出的程序除了在初始化预测分析表时做出适当的修改外，其他的程序内容无需改变，所以为了写出的代码在今后可以得到重复使用，即使修改文法后也可以不做很多的修改内容也能实现自上而下的语法分析任务，本次实验选择了使用 **预测分析法** 实现。对于预测分析法，最重要的有两点，一是预测分析表的构造，另一个就是总控程序的编写，对于后者，总控程序的基本算法逻辑是一定的，所以重点是前期要将想识别的文法的预测分析表构造出来即可。除此之外，在具体的实现过程中，为了简化某些操作以及使得代码的可读性更强，所以要先实现一些基础实体对应的类，例如基础符号类、非终结符类、终结符类、产生式类以及预测分析表类等等，因为在之前的课程学习中，编写面向对象的工程大多使用的是Java，所以在这次使用cpp来实现面向对象的代码是对我的一个复习，不仅从这次实现掌握了编译原理这门课的自上而下语法分析的相关内容，而且复习了之前学习中遗忘的知识。此外这次试验也给我很大启示，编写代码时，尤其是较大的工程，一定要先将思路理清，确定需要的类以及相互的关系，不能像以前那样先上手敲一些，然后停下来想一会再继续敲，这样写出的代码一定会有一些问题，当全部相通后，用一两个小时便可编写完所有代码并调试成功。对于预测分析法的感受是，预测分析表可以看成一个关于各产生式之间转化的一个图的关系，显然当某一个格子存在产生式就意味着存在一条边，而一个句子符合某个文法的语法结构，就是能在这张图中从起点（文法开始符）到终点（句末符#）找到一条路径，总控程序就是一个图转移的过程，也就是将一个递归搜索的过程改为了循环结构，并用栈模拟。

HTML