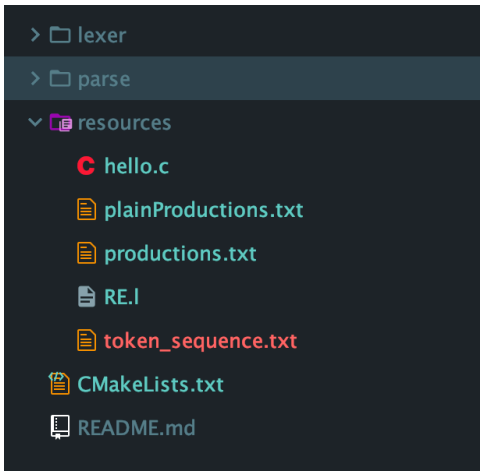
编译原理实验报告

姓名： 曾少勋 学号：171250603

项目结构：



输入文件：

```
functions -> function functions1
functions1 -> @
functions1 -> functions
function -> ret ID ( ) { states }
ret -> VOID
ret -> INT
ret -> FLOAT
ret -> DOUBLE
states -> state states1
states1 -> @
states1 -> states
state -> init ;
state -> RETURN RTYPE ;
RTYPE -> @
RTYPE -> ID
RTYPE -> INTEGER
RTYPE -> FLOAT_NUM
RTYPE -> DOUBLE
TYPE -> INT
TYPE -> FLOAT
TYPE -> DOUBLE
init -> TYPE ID init1
init1 -> @
init1 -> = init2
init2 -> arithmetic
operand -> ID
operand -> INTEGER
operand -> FLOAT_NUM
arithmetic -> operand arithmetic1
arithmetic1 -> @
arithmetic1 -> op operand
```

核心数据结构：

```cpp
typedef struct production {
    // 对应产生式的左部
    string Vn;
    // 产生式的右部
    vector<string> items;

    friend ostream &operator<<(ostream &os, production &p);

    // 自定义类型需要重载<使set中的值唯一
    int operator<(const production &p) const {
        if (Vn == p.Vn) {
            return items < p.items;
        }
        return Vn < p.Vn;
    }
} Production;
```

```
class ALLProductions {
private:
    string start;
    set<string> Vns;
    set<Production> allProductions;
    map<string, set<Production>> sameVnProductions;

    void add(Production production);

    set<string> follow(string Vn, set<string> &visited);

    set<string> first(vector<string> items, set<vector<string>> &visited);

public:
    set<string> first(vector<string> items);

    set<string> follow(string Vn);

    bool isVn(string item);

    void add(string production);

    friend void first_and_follow();
};
```

```
class PPT{
private:
    // 第一个string为Vn, 第二个string为Vt, Production为产生式
    map<string, map<string, Production>> table;
public:
    void add(Production p, string Vt);
    Production search(string Vn, string Vt);
    bool isVn(string s);
    friend void printPPT(const PPT &ppt);
};
```

核心算法：

```
while (!items.empty()) {
    // 取出序列中的第一个求first
    set<string> tres = first(items, &: visited);
    // 如果结果中不包含epsilon边，则退出循环
    if (tres.find( k: "@") == tres.end()) {
        res.insert(tres.begin(), tres.end());
        break;
    }
        // 否则将除了epsilon之外的first集合元素加入结果集合中
        // 并对下一个item求first
    else {
        tres.erase( k: "@");
        res.insert(tres.begin(), tres.end());
        items.erase( position: items.begin());
    }
    // 直到序列为空
}
// 如果items为空，则说明该产生式中每一个item的first都包含epsilon
// 此时需要将epsilon加入结果集中
if (items.empty()) {
    res.insert( v: "@");
}
```

```
// 如果first中包含epsilon边，那么还要求follow
if (firsts.find( k: "@") != firsts.end()) {
    set<string> follows = allProductions.follow(p.Vn);
    firsts.insert(follows.begin(), follows.end());
}
```

```
// 如果包含该非终结符
vector<string>::iterator pos;
if ((pos = find(production.items.begin(), production.items.end(), Vn)) != production.items.end()) {
    // 该非终结符还不在产生式的末尾
    if (pos != production.items.end() - 1) {
        vector<string> items;
        items.assign( first: pos + 1, production.items.end());
        set<string> firsts = this->first(items);
        // 如果没有epsilon
        if (firsts.find( k: "@") == firsts.end()) {
            res.insert(firsts.begin(), firsts.end());
            // 只有这种情况才会跳过后面的流程
            continue;
        } else {
            // 如果有epsilon，那么将除了epsilon之外的边加入结果中
            // 并且还要继续求解
            firsts.erase( k: "@");
            res.insert(firsts.begin(), firsts.end());
        }
    }
    // 将该产生式的follow加入结果集中
    set<string> follows = this->follow(production.Vn,  &: visited);
    res.insert(follows.begin(), follows.end());
```

```
void PPT::add(Production p, string Vt){
    if(Vt == "@") return;
    string Vn = p.Vn;
    if(this->table.find(Vn) == this->table.end()){
        map<string, Production> t;
        t.emplace(Vt, p);
        this->table.emplace(Vn, t);
    } else {
        this->table[Vn].emplace(Vt, p);
    }
}
```

```
while ((pos = Vts.find( c: ' ',  pos: former + 1)) != string::npos) {
    ppt.add(p,  Vt: Vts.substr( pos: former + 1,  n: pos - former - 1));
    former = pos;
}
```

```
queue<string> readers = read_token_sequence( filename: "resources/token_sequence.txt");
readers.push( v: "$");
```

```
// 栈顶为非终结符
if (ppt.isVn(stack.top())) {
    string Vn = stack.top();
    stack.pop();
    Production p = ppt.search(Vn, Vt);
    // 如果没有找到对应的产生式则报错
    if (p.Vn.empty()) {
        error = true;
        cout << "error" << endl;
        break;
    } else {
        productions.push_back(p);
        // 将产生式反着压入栈中
        for (auto it = p.items.rbegin(); it != p.items.rend(); it++) {
            stack.push(*it);
        }
    }
}
```

```
        // 栈顶为终结符
    else {
        // 如果非终结符一样，说明匹配
        if (stack.top() == Vt) {
            stack.pop();
            Vt = readers.front();
            readers.pop();
        }
            // 如果栈顶是epsilon，那么直接弹栈
        else if (stack.top() == "@") {
            stack.pop();
        }
            // 否则说明出现错误
        else {
            error = true;
            cout << "error!!!" << endl;
            break;
        }
    }
}
```

输入的 c 源文件

C 源文件：

```
void f(){
    return;
}

int main(){
    int complex_AND_long_variable_name_100 = 1 + 1;
    double y = 1.0 * 2.0;
    if(x < y){
        x = y;
    } else {
        y = x;
    }
    int i = 0;
    while(i < 10){
        i = i + 1;
    }
    return 0;
}
```

控制台输出推导产生式序列（因为比较长，前面一部分截图如下）：

```
functions -> function functions1
function -> ret ID ( ) { states }
ret -> VOID
states -> state states1
state -> RETURN RTYPE ;
RTYPE -> @
states1 -> @
functions1 -> functions
functions -> function functions1
function -> ret ID ( ) { states }
ret -> INT
states -> state states1
state -> init ;
init -> TYPE ID init1
TYPE -> INT
init1 -> = init2
init2 -> arithmetic
arithmetic -> operand arithmetic1
operand -> INTEGER
arithmetic1 -> op operand
op -> +
operand -> INTEGER
states1 -> states
states -> state states1
state -> init ;
init -> TYPE ID init1
TYPE -> DOUBLE
```

Motivation/Aim

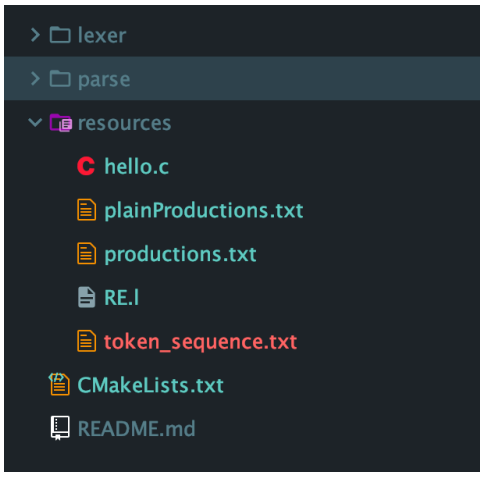
用程序实现 LL(1)算法，进行语法分析

输入为一个 c 源文件，通过上次实验的词法分析生成 token 序列，将 token 序列输入语法分析程序，输出推导的产生式序列

Content description

项目使用 CMake 构建，将整个工程导入 Clion 可直接运行，或是用其他 CMake 构建工具构建运行

项目结构为：



其中 lexer 为词法分析程序

parse 为语法分析程序

resources 中为输入文件和中间结果文件，plainProductions.txt 存放用户定义的语法

parse 中文件结构如下：

需要先运行 lexer 中的 Lexer.cpp，然后再运行 parse 中的 Yacc.cpp，即可在控制台输出推导的产生式序列

用户定义的文法，因为比较长，截图选取了一部分(文件为 plainProductions.txt)：

```
functions -> function functions1
functions1 -> @
functions1 -> functions
function -> ret ID ( ) { states }
ret -> VOID
ret -> INT
ret -> FLOAT
ret -> DOUBLE
states -> state states1
states1 -> @
states1 -> states
state -> init ;
state -> RETURN RTYPE ;
RTYPE -> @
RTYPE -> ID
RTYPE -> INTEGER
RTYPE -> FLOAT_NUM
RTYPE -> DOUBLE
TYPE -> INT
TYPE -> FLOAT
TYPE -> DOUBLE
init -> TYPE ID init1
init1 -> @
init1 -> = init2
init2 -> arithmetic
operand -> ID
operand -> INTEGER
operand -> FLOAT_NUM
arithmetic -> operand arithmetic1
arithmetic1 -> @
arithmetic1 -> op operand
```

Ideas/Methods

采用 LL(1)算法，全部过程使用程序实现，详见核心算法描述

assumption

假设定义的文法已经消除了左递归和提取了公共左因子，且无二义性，是 LL(1)文法

假设前一次实验的 Lexer 实现是正确的

重要的数据结构描述

定义了产生式的结构，用于读取和转化用户定义的语法：

```
typedef struct production {
    // 对应产生式的左部
    string Vn;
    // 产生式的右部
    vector<string> items;

    friend ostream &operator<<(ostream &os, production &p);

    // 自定义类型需要重载<使set中的值唯一
    int operator<(const production &p) const {
        if (Vn == p.Vn) {
            return items < p.items;
        }
        return Vn < p.Vn;
    }
} Production;
```

为了便于统一管理产生式，专门定义了一个容器来存放所有产生式，并将 first 和 follow 定义为类中的方法：

```
class ALLProductions {
private:
    string start;
    set<string> Vns;
    set<Production> allProductions;
    map<string, set<Production>> sameVnProductions;

    void add(Production production);

    set<string> follow(string Vn, set<string> &visited);

    set<string> first(vector<string> items, set<vector<string>> &visited);

public:
    set<string> first(vector<string> items);

    set<string> follow(string Vn);

    bool isVn(string item);

    void add(string production);

    friend void first_and_follow();
};
```

PPT 表格数据结构，用于存储 PPT：

```
class PPT{
private:
    // 第一个string为Vn，第二个string为Vt，Production为产生式
    map<string, map<string, Production>> table;
public:
    void add(Production p, string Vt);
    Production search(string Vn, string Vt);
    bool isVn(string s);
    friend void printPPT(const PPT &ppt);
};
```

核心算法描述

首先读取用户的文法文件，构造 Production

之后求出 first 和 follow（如果需要的话）

求 first 的算法采用递归，核心的步骤为：

```
while (!items.empty()) {
    // 取出序列中的第一个求first
    set<string> tres = first(items, &: visited);
    // 如果结果中不包含epsilon边，则退出循环
    if (tres.find( k: "@") == tres.end()) {
        res.insert(tres.begin(), tres.end());
        break;
    }
        // 否则将除了epsilon之外的first集合元素加入结果集合中
        // 并对下一个item求first
    else {
        tres.erase( k: "@");
        res.insert(tres.begin(), tres.end());
        items.erase( position: items.begin());
    }
    // 直到序列为空
}
// 如果items为空，则说明该产生式中每一个item的first都包含epsilon
// 此时需要将epsilon加入结果集中
if (items.empty()) {
    res.insert( v: "@");
}
```

求出 first 之后，判断是否包含 e 边，如果有的话，还要进一步求 follow：

```
// 如果first中包含epsilon边，那么还要求follow
if (firsts.find( k: "@") != firsts.end()) {
    set<string> follows = allProductions.follow(p.Vn);
    firsts.insert(follows.begin(), follows.end());
}
```

求 follow 的核心算法如下：

遍历所有的产生式右部，如果包含该非终结符，判断位置，根据位置分情况讨论

```
// 如果包含该非终结符
vector<string>::iterator pos;
if ((pos = find(production.items.begin(), production.items.end(), Vn)) != production.items.end()) {
    // 该非终结符还不在产生式的末尾
    if (pos != production.items.end() - 1) {
        vector<string> items;
        items.assign( first: pos + 1, production.items.end());
        set<string> firsts = this->first(items);
        // 如果没有epsilon
        if (firsts.find( k: "@") == firsts.end()) {
            res.insert(firsts.begin(), firsts.end());
            // 只有这种情况才会跳过后面的流程
            continue;
        } else {
            // 如果有epsilon，那么将除了epsilon之外的边加入结果中
            // 并且还要继续求解
            firsts.erase( k: "@");
            res.insert(firsts.begin(), firsts.end());
        }
    }
    // 将该产生式的follow加入结果集中
    set<string> follows = this->follow(production.Vn,  &: visited);
    res.insert(follows.begin(), follows.end());
```

求出所有表达式的 first 以及需要的 follow 之后，将中间结果全部输出到一个临时文件(productions.txt)中，这样不仅可以方便 debug，还可以让求 first 和 follow 的过程和构造 ppt 的过程解耦，有利于模块和方法的复用

接下来就是构造 PPT

构造 ppt 分为两个步骤，为了便于模块化和复用，利用了面向对象的机制，在 ppt 类中封装了添加一个表项的方法，输入一个 Production，即可添加一个表项：

```
void PPT::add(Production p, string Vt){
    if(Vt == "@") return;
    string Vn = p.Vn;
    if(this->table.find(Vn) == this->table.end()){
        map<string, Production> t;
        t.emplace(Vt, p);
        this->table.emplace(Vn, t);
    } else {
        this->table[Vn].emplace(Vt, p);
    }
}
```

然后在一个循环中遍历刚刚生成的中间结果文件（productions.txt），构造产生式，然后调用 ppt 中的 add 方法，即可构造表格

核心的步骤如下：

```
while ((pos = Vts.find( c: ' ',  pos: former + 1)) != string::npos) {
    ppt.add(p,  Vt: Vts.substr( pos: former + 1,  n: pos - former - 1));
    former = pos;
}
```

ppt 构造完成之后，读取词法分析产生的 token 序列：

```
queue<string> readers = read_token_sequence( filename: "resources/token_sequence.txt");
readers.push( v: "$");
```

接下来模拟 monitor 的过程，维护栈和队列，进行语法分析和产生式序列的构造：

根据栈顶是终结符或是非终结符，有：

```
// 栈顶为非终结符
if (ppt.isVn(stack.top())) {
    string Vn = stack.top();
    stack.pop();
    Production p = ppt.search(Vn, Vt);
    // 如果没有找到对应的产生式则报错
    if (p.Vn.empty()) {
        error = true;
        cout << "error" << endl;
        break;
    } else {
        productions.push_back(p);
        // 将产生式反着压入栈中
        for (auto it = p.items.rbegin(); it != p.items.rend(); it++) {
            stack.push(*it);
        }
    }
}
```

```
        // 栈顶为终结符
    else {
        // 如果非终结符一样，说明匹配
        if (stack.top() == Vt) {
            stack.pop();
            Vt = readers.front();
            readers.pop();
        }
            // 如果栈顶是epsilon，那么直接弹栈
        else if (stack.top() == "@") {
            stack.pop();
        }
            // 否则说明出现错误
        else {
            error = true;
            cout << "error!!!" << endl;
            break;
        }
    }
}
```

整个语法分析过程到此结束

Example

输入定义语法文件：

```
functions -> function functions1
functions1 -> @
functions1 -> functions
function -> ret ID ( ) { states }
ret -> VOID
ret -> INT
ret -> FLOAT
ret -> DOUBLE
states -> state states1
states1 -> @
states1 -> states
state -> init ;
state -> RETURN RTYPE ;
RTYPE -> @
RTYPE -> ID
RTYPE -> INTEGER
RTYPE -> FLOAT_NUM
RTYPE -> DOUBLE
TYPE -> INT
TYPE -> FLOAT
TYPE -> DOUBLE
init -> TYPE ID init1
init1 -> @
init1 -> = init2
init2 -> arithmetic
operand -> ID
operand -> INTEGER
operand -> FLOAT_NUM
arithmetic -> operand arithmetic1
arithmetic1 -> @
arithmetic1 -> op operand
```

C 源文件：

```c
void f(){
    return;
}

int main(){
    int complex_AND_long_variable_name_100 = 1 + 1;
    double y = 1.0 * 2.0;
    if(x < y){
        x = y;
    } else {
        y = x;
    }
    int i = 0;
    while(i < 10){
        i = i + 1;
    }
    return 0;
}
```

控制台输出推导产生式序列（因为比较长，前面一部分截图如下）：

```
functions -> function functions1
function -> ret ID ( ) { states }
ret -> VOID
states -> state states1
state -> RETURN RTYPE ;
RTYPE -> @
states1 -> @
functions1 -> functions
functions -> function functions1
function -> ret ID ( ) { states }
ret -> INT
states -> state states1
state -> init ;
init -> TYPE ID init1
TYPE -> INT
init1 -> = init2
init2 -> arithmetic
arithmetic -> operand arithmetic1
operand -> INTEGER
arithmetic1 -> op operand
op -> +
operand -> INTEGER
states1 -> states
states -> state states1
state -> init ;
init -> TYPE ID init1
TYPE -> DOUBLE
```

Problems occurred and related solutions

实验中遇到许多问题，举几个经典的问题如下：

1. 读取文件转为产生式的问题.

   这里因为考虑到用户体验，采用 ->加上空格的形式定义产生式，转化成对应的 Production 类型的时候遇到了一些问题，最后是通过不断的调试解决

2. 记忆化搜索带来的负面问题

   在求解 first 和 follow 的时候，因为 first 和 follow 容易产生循环求解同一个 first 和 follow 的现象，因此考虑使用记忆化搜索，即在 first 和 follow 中每次都保存下当前求解的值，以便下次直接查找。原以为这样会提高效率，但是实际上带来了 bug。考虑如下情况：

   First(A) = First(A) + Follow(B)

   而求解 Follow(B)的时候可能又会求解到 First(A)，这里因为第二次搜索到了 A，为了避免死循环，会直接让这个 First(A)返回一个空集合，然后 Follow(B)会认为已经求解过了，会记录下来，下一次用到 Follow(B)就直接返回一个空集，然而实际上这是不对的，这里的 First(A)返回空集只是因为是第二次求解到 A，而单独求解 Follow(B)不应该是空集

   因此为了程序的正确性，只能删除掉了记忆化搜索部分

3. 二义文法带来的问题

   在构造文法的时候，为了能使用 LL(1)进行分析，不能含有左递归和公共左因子，考虑如下文法：

   A -> B c

   B -> e（其中 e 为 epsilon 边）

   B -> c

表面上看，A 和 B 并没有公共左因子，然而因为 B 可能会推导出 epsilon 边，因此实际上 A 的 first 中是包含 c 的，而 B 的 first 中也包含 c，这样构造出来的 LL 分析表实际上是有冲突的

为什么看起来已经消除了公共左因子归却仍然有冲突呢？

仔细分析可以发现，实际上面定义的文法是二义文法，例如对于 c 来说，就可能有两种语法树

因为自定义的语法序列比较长，因此难免出现上面这种看起来没有公共左因子但是实际上是二义文法的情况，经过全方位的仔细排查才消除了二义文法

Your feelings and comments

通过实验，进一步理解了 LL(1)文法的特点，以及运用 LL(1)来进行语法分析的过程

通过大量的调试过程，更加理解了为什么要在 coding 之前先做细致的分析和思考，例如上面的二义文法，就是因为刚开始没有考虑清楚导致的

体验到了细心和耐心在整个项目的过程中是至关重要的