# 《词法分析程序的设计与实现》

# 实验报告

**姓名：于孟孟**

**学号：2022211260**

**班级：2022211307**

**2024年12月6日**

[《词法分析程序的设计与实现》 1](#_Toc184502304)

[实验报告 1](#_Toc184502305)

[一、 LL(1)语法分析程序 3](#_Toc184502306)

[（一） 实验题目 3](#_Toc184502307)

[（二） 程序设计说明 6](#_Toc184502308)

[（三） 源程序 14](#_Toc184502309)

[（四） 可执行程序 14](#_Toc184502310)

[（五） 测试报告 15](#_Toc184502311)

[二、 LR(1)语法分析程序 16](#_Toc184502312)

[（一） 实验题目 16](#_Toc184502313)

[（二） 程序设计说明 18](#_Toc184502314)

[（三） 源程序 28](#_Toc184502315)

[（四）可执行程序 28](#_Toc184502316)

[（五）测试报告 29](#_Toc184502317)

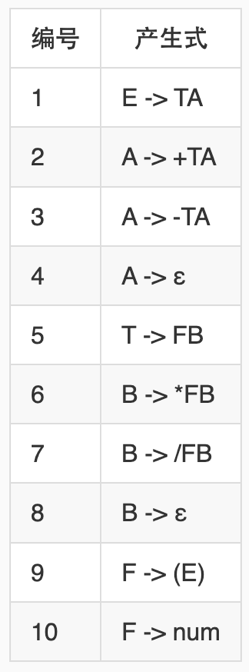
## LL(1)语法分析程序

### 实验题目

#### 任务描述

编写一个LL(1)语法分析程序，能对算术表达式进行语法分析。

要求：在如下消除左递归的文法G’的基础上



1. 编程实现，为给定文法自动构造预测分析表；
2. 编程实现，为LL(1)构造预测分析程序；

#### 编程要求

（1）使用C/C++实现。

（2）头歌平台环境：

编译器版本 gcc7.3.0

OS版本 Debian GNU/Linux 9

#### 测试说明

1. 输入格式

<1> 从标准输入(cin/scanf)读入数据；

<2> 输入仅包含一行：一个算术表达式，构成该算术表达式的字符有：{ ‘n’, ‘+’, ‘-‘, ‘\*’, ‘/’, ‘(‘, ‘)’ }

1. 输出格式

<1> 输出到标准输出(cout/printf)中;

<2> 输出包含若干行LL(1)分析过程，假设输出有n行，则第i行(1 ≤ i ≤ n)表示分析进行到第i步，它的输出包括如下三个部分：

1. 分析栈：以$符号表示栈底的字符串(左侧表示栈底，由终结符和非终结符构成)；
2. 输入串栈：以$符号表示栈底的字符串(右侧为栈底)；
3. 分析动作：产生式编号，或match或error或accept(表示当前步骤应执行的动作)。
4. 样例输入与输出

<1> 测试集1

测试输入：n + n

测试输出：

$E n+n$ 1

$AT n+n$ 5

$ABF n+n$ 10

$ABn n+n$ match

$AB +n$ 8

$A +n$ 2

$AT+ +n$ match

$AT n$ 5

$ABF n$ 10

$ABn n$ match

$AB $ 8

$A $ 4

<2> 测试集2

测试输入：n-n\*n

测试输出：

$E n-n\*n$ 1

$AT n-n\*n$ 5

$ABF n-n\*n$ 10

$ABn n-n\*n$ match

$AB -n\*n$ 8

$A -n\*n$ 3

$AT- -n\*n$ match

$AT n\*n$ 5

$ABF n\*n$ 10

$ABn n\*n$ match

$AB \*n$ 6

$ABF\* \*n$ match

$ABF n$ 10

$ABn n$ match

$AB $ 8

$A $ 4

$ $ accept

### 程序设计说明

#### 概要设计

##### 模块划分



##### 1.2 各模块说明

1. table的构造函数
   1. 构建start，文法的起始符；
   2. 构建notends，文法的非终结符；
   3. 构建characters，文法的终结符，用n代表num，e代表空；
   4. 将所有产生式存入table中，作为generator；
   5. 初始化buildedfirst(非终结符是否已经构造过first集合的标志，开始时全为false)；
   6. 初始化buildedfollow(非终结符是否已经构造过follow集合的标志，开始时全为false)；
   7. 初始化reachempty集合，将所有能产生空串的字符存入set中；
   8. 初始化passfollow集合，该集合将在构造follow集合的时候用于标记follow集合的传递关系是否已经被使用过，避免重复、循环传递；
   9. 初始化analyzestack和inputcharstack，分析栈和符号栈，将$和文法起始符压入分析栈，将$压入输入符号栈(其他输入符号会在analyzeinput()中压入)
2. buildfirstset()

为每个非终结符调用findfirst函数，构建first集合；

1. buildfollowset()
   1. 将$符加入起始符的follow集合；
   2. 遍历所有非终结符，构造其显示的follow集合，同时记录不同非终结符的follow集合之间的传递关系；
   3. 遍历follow集合之间的传递关系，将follow集合传递，通过查passfollow表来保证不重复；
   4. 去掉follow集合中的e，即空产生式；
2. buildtable()
   1. 遍历所有产生式，根据之前构造的first集合和follow集合，利用如下算法构造预测分析表

for (文法G的每个产生式A->å) {

for (每个终结符号a 属于FIRST(å))

把A->å放入M[A, a]中；

if (e 属于FIRST(å))

for (任何b属于FOLLOW(A)

把A -> å 放入M[A, b]中；

};

1. analyzeinput()
   1. 将标准输入压入输入符号栈；
   2. 从分析栈中取出left，从符号栈中取出c,
      1. 若left与c相等，则match；
      2. 若left与c不相等且left是终结符，在M表中查找M[left, c], 找到对应的产生式，打印产生式编号，根据产生式对分析栈进行操作，产生式左部部元素出栈(分析栈顶部元素)，产生式右部元素入栈；如果M[left, c]为空，则error；

##### 数据结构

class Table中的private成员存储所有的数据

class Table{

public:

成员函数

private:

vector<char> characters; // 终结符号

vector<char> notends; // 非终结符号

set<char> reachempty; // 空产生式

map<char, vector<vector<char> > > generator; // 产生式表

map<char, set<char> > firstset; // first集合

map<char, set<char> > followset; // follow集合

vector<char> buildedfirst; // 记录该符号是否已经构造过first集合

vector<char> buildedfollow; // 记录该符号是否已经构造过follow集合

char start; // 文法的起始符号

map<char, vector<char>> son; // 记录follow集合的依赖关系

vector<bool> passfollow; // 记录此依赖关系是否已经传递过follow集合，避免重复传递

map <char, map<char, vector<char> > > M; // 分析表

map <char, map<char, int> > id; // 分析表中产生式对应的id

vector<char> analyzestack; // 分析栈

vector<char> inputcharstack; // 符号栈

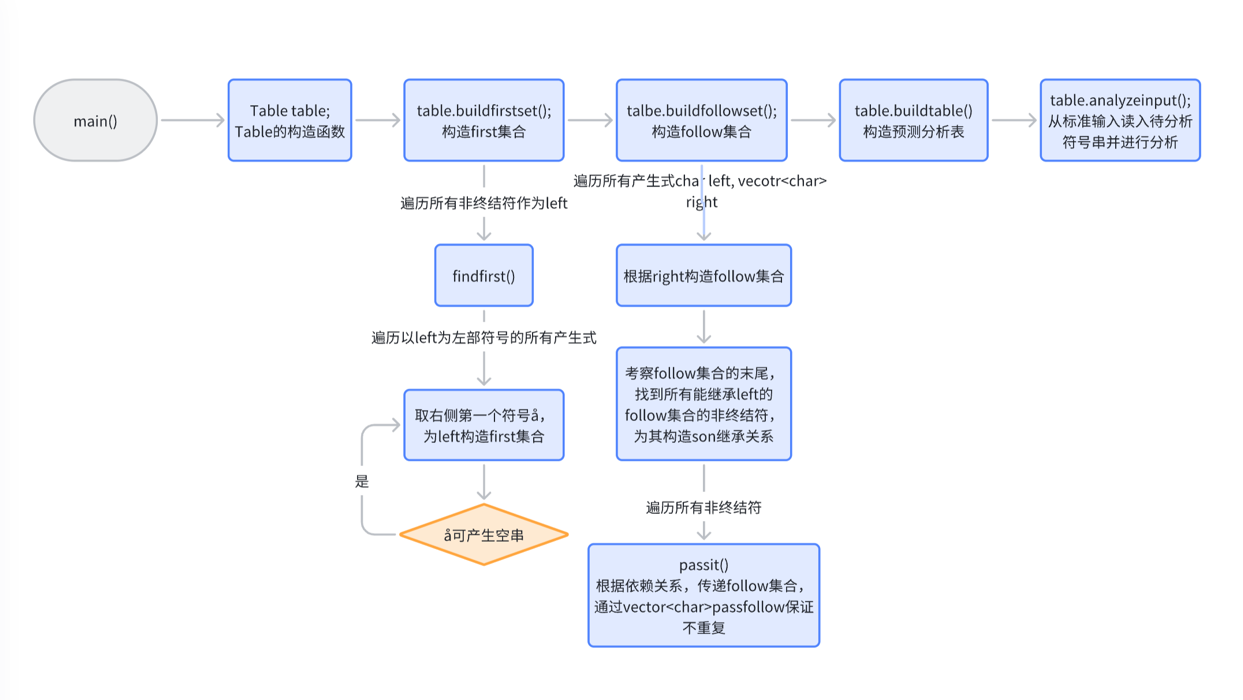
};

1. 文法起始符

char start

1. 终结符表&非终结符表
   1. vector<char> characters; 终结符表
   2. vector<char> notends; 非终结符表
2. 产生式表
   1. map<char, vector<vector<char>>> generator;
   2. 主键代表产生式左部符号；
   3. vector<vector<char>> 代表产生式左部符号相同的所有产生式右部；
   4. 最里层的vector<char>存储了一个产生式右部的所有符号；
3. first集&follow集合
   1. map<char, set<char>> firstset; 主键代表一个非终结符，set<char>表示非终结符对应的first集合；
   2. map<char, set<char>> followset; 主键代表一个非终结符，set<char>表示非终结符对应的follow集合；
4. 分析表
   1. map<char, map<char, vector<char>>> M;
   2. 第一层主键代表非终结符；
   3. 第二层主键代表终结符；
   4. 最内层代表需要使用的产生式的右部符号；
   5. 使用：在分析输入符号时，遇到的输入符号一定是终结符， 当分析栈中为非终结符时，查找M表，找到对应的产生式右部符号，将分析栈栈顶出栈，产生式右部符号入栈，并打印产生式编号(通过returnid()函数，根据产生式内容返回其编号)；
5. 分析栈&符号栈
   1. vector<char> analyzestack;
   2. vector<char> inputcharstack;
   3. push\_back()入栈，pop\_back()出栈
6. id表
   1. map<char, map<char, int>>id；
   2. 与分析表对应，表示相应产生式对应的编号，便于分析过程的输出；

#### 详细设计



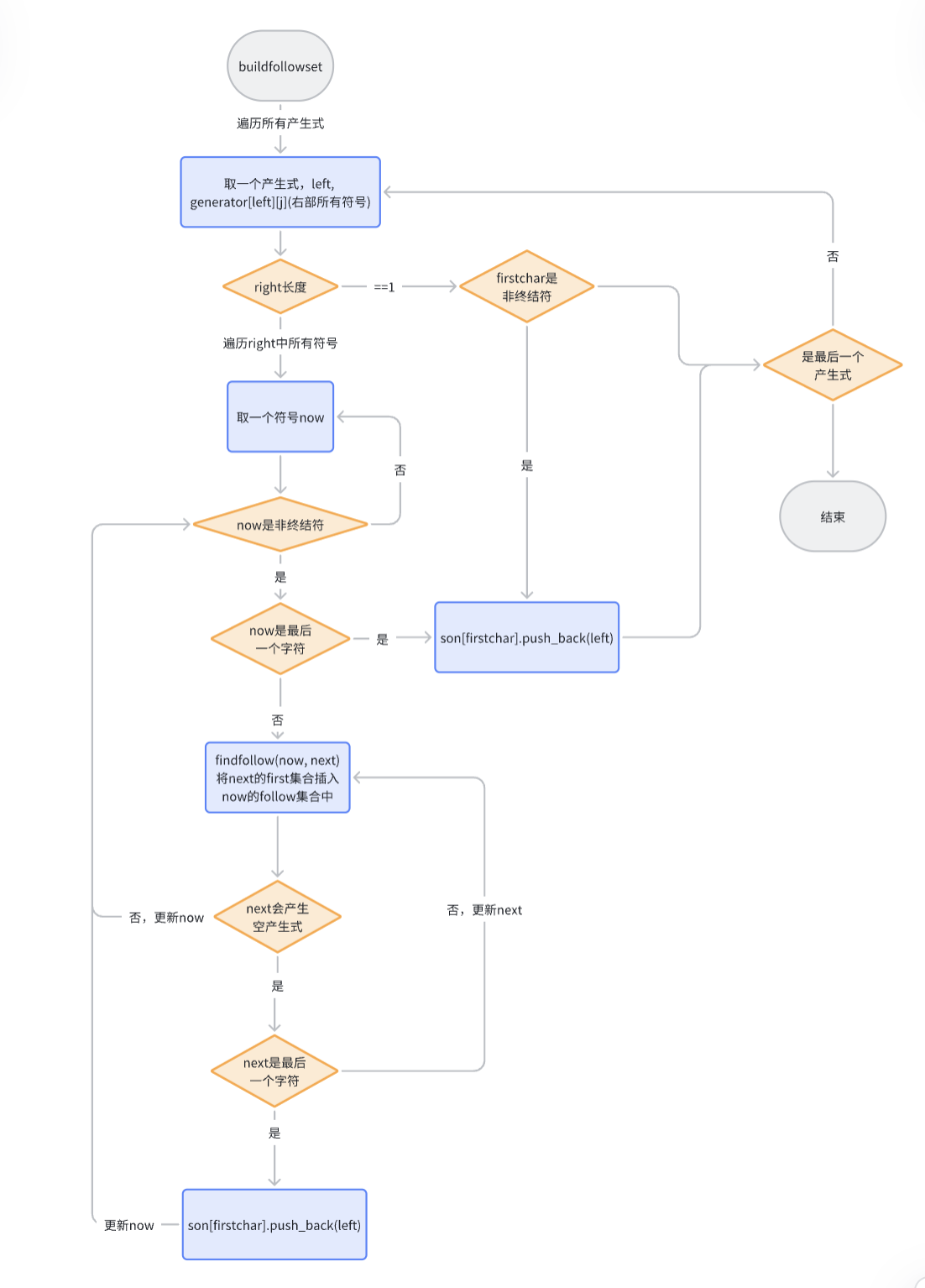
##### first集合的构造

图示

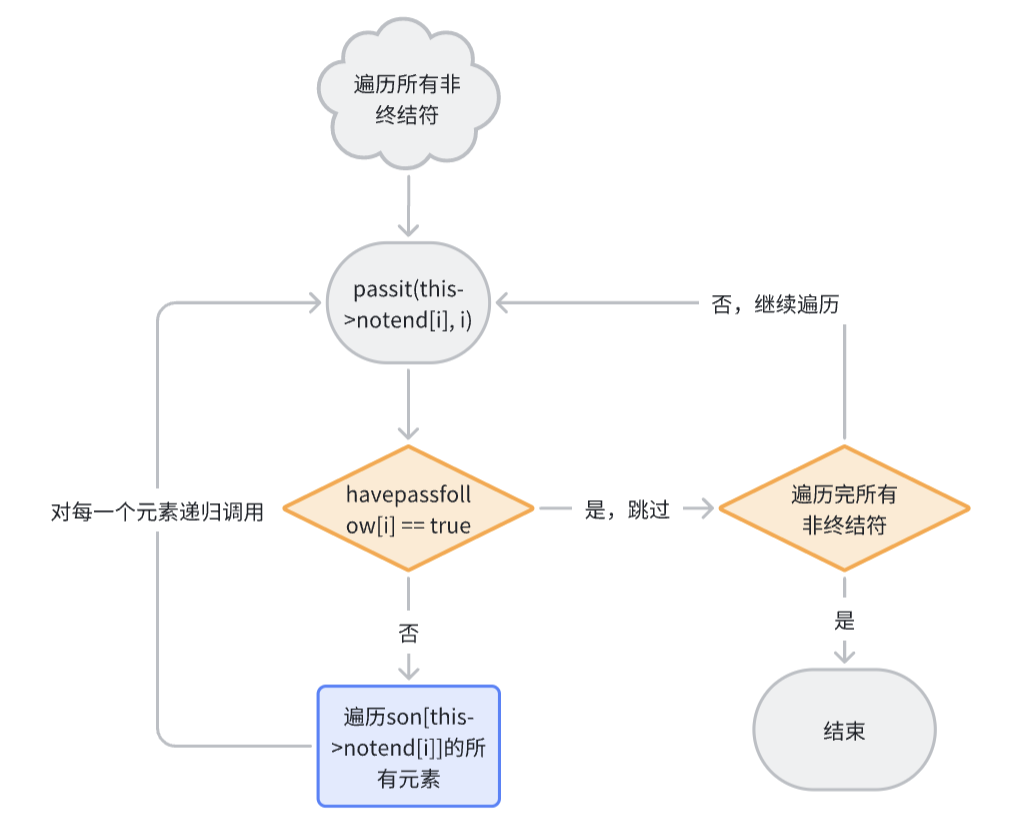
描述已自动生成

##### follow集合的构造

(1) 第一步直接构造



（2）follow集合的传递；



（3）去掉所有follow集合中的空串；

##### 分析表的构造图示 描述已自动生成

##### 分析程序

图示

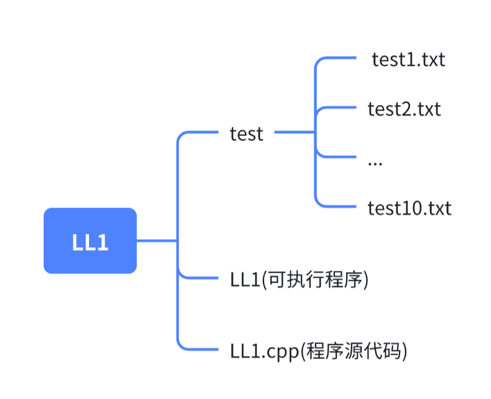
描述已自动生成

##### 过程输出(<<运算符重载)

输出包括start, Charaters, notends, Generator, Firstset, Followset, Table(Mtable), Id(Id table)

### 源程序

##### 提交文件结构如下：



##### 3.2 编译指令

* 1. 在LL1/下
  2. g++ LL1.cpp -o LL1 -std=c++11

##### 3.3 中间信息输出

将源代码622行cout << table;的注释取消，即可打印2.5中提到的过程输出；

### 可执行程序

1. LL1/LL1
2. 执行方式
3. ./LL1 <test/test1.txt
4. ./LL1 程序运行后从标准输入输入一行待分析的字符串

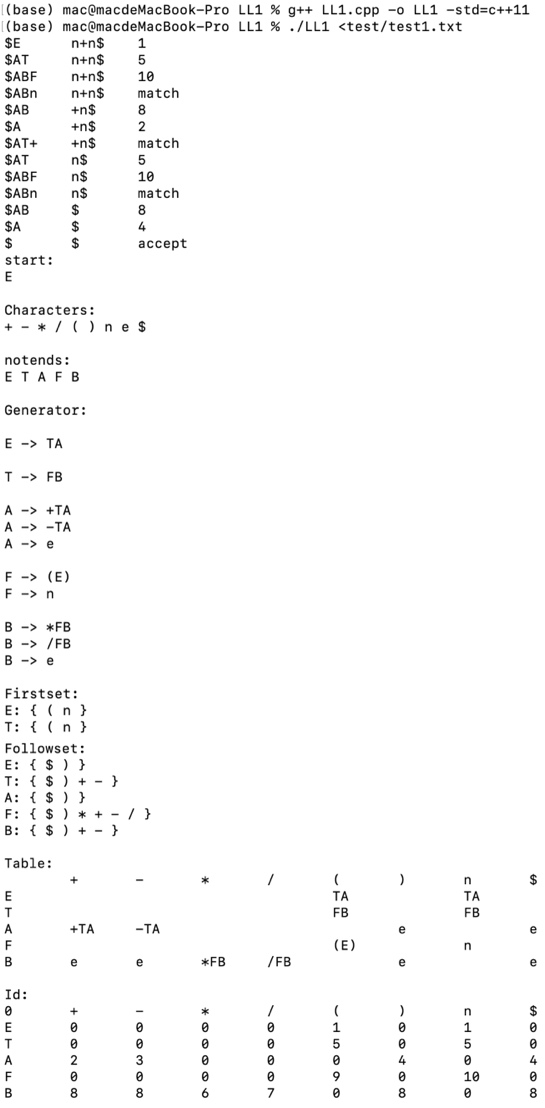
### 测试报告

##### 5.1 头歌平台测试



##### 5.2 本地测试

1. test1.txt



1. test2.txt

图片包含 表格

描述已自动生成

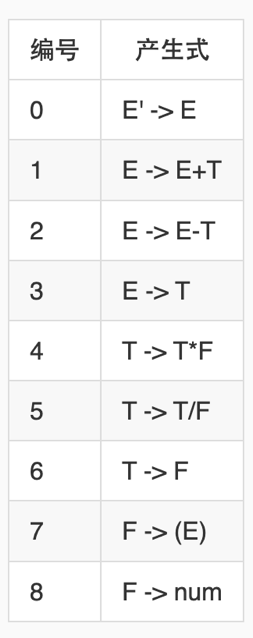
## LR(1)语法分析程序

### 实验题目

#### 1 任务描述

编写一个语法分析程序，能对算术表达式进行LR(1)语法分析。

要求：



（1）编程实现，构造该文法的LR(1)分析表；

（2）编程实现算法4.3，构造LR(1)分析程序；

#### 2. 编程要求

（1）使用C/C++实现

（2）平台环境说明

编译器版本：gcc7.3.0

OS版本：Debian GNU/Linux 9

#### 3. 测试说明

（1）输入格式：

<1> 从标准输入(cin/scanf)读入数据；

<2> 输入仅包含一行，为一个算术表达式，由{‘n’, ‘+’, ‘-‘, ‘\*’, ‘/’, ‘(‘, ‘)’}构成；

（2）输出格式：

<1> 输出到标准输出(cout/printf等输出)中；

<2> 输出包括若干行LR(1)分析过程，共包含n行；

<3> 第i行表示分析进行到第i步是所采取的动作：

a. 归约使用的产生式编号；

b. shift/error/accept

（3）样例输入与输出

<1> 测试集1

a. 输入 n+n

b. 输出

shift

8

6

3

shift

shift

8

6

1

accept

### 程序设计说明

#### 1. 概要设计

##### 1.1 模块划分



##### 1.2 各模块说明

(1) Table构造函数

<1> 初始化文件起始符，用S代替E’表示起始符；

<2> 初始化非终结符数组；

<3> 初始化非终结符数组，用e表示空，并额外存入’$’；

<4> 将产生式存入generator；

<5> 初始化buildedfirst，buildedfollow，用于记录是否每一个非终结符是否已经构建过first集合和follow集合；

<6> 初始化havepassfollow，用于记录所有follow集合的传递关系是否已经使用过，避免重复传递；

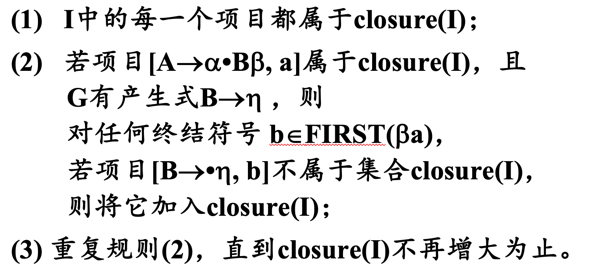
(2) buildfirstset() & buildfollowset()函数；

与LL(1)文法完全相同，见一；

(4) builditemset()函数；

构造LR1项目集；

<1> 将S->·E, $存入states[0]中，按如下算法构造闭包，形成状态0；



<2> 从状态0出发，遍历所有可读取的字符，构建新的状态，并填充goto表(jumptable)；

(5) buildRtable()

<1>遍历所有状态，找到所有具有归约串的状态，填充Rtable；

<2>需要查看jumptable中对应位置有没有动作，如果有的话，则产生移进-归约冲突，该文法不是LR(1)文法

<3>默认没有归约-归约冲突

(6) readstring() & analyzestring()函数

<1> 将$符压入inputstack，从标准输入读入待分析串，压入inputstack；

<2> 将状态0压入statestack，将’$’压入sigstack;

<3> 根据当前statestack和inputstack的栈顶符号查找jumptable

a. 如果对应的jumptable表项不为空，则执行状态跳转，shift，statestack和sigstack入栈，inputstack出栈；

b. 如果对应的jumptable表项为空，则查找Rtable

<a>. 如果对应的Rtable表项不为空，则按照Rtable中存储的内容，进行归约，将用于归约用的产生式右侧对应的statestack和sigstack出栈，再将产生式左侧字符left作为当前输入符号，循环执行<3>；

<b>如果对应的Rtable表项为空，则error;

##### 1.3 数据结构

class Table中存储所有数据；其中有自定义数据类型Content

typedef struct content // 有效项目的右侧内容

{

vector<char> readed; // 归约串

vector<char> waitread; // 待约串(倒序)

set<char> lookforward; // 向前看符号

bool operator==(const content& other) const { }

// ==运算符重载，当readed与waitread相等时，视为两个content相等

bool operator<(const content& other) const { }

// 先按readed.size()，再按waitreaded.size(), 再按readed的内容，再按waitreaded的内容排序

}Content;

class Table

{

public:

成员函数

private:

vector<char> endsig; // 终结符号

vector<char> notend; // 非终结符号

set<char> reachempty; // 空产生式

map<char, vector<vector<char> > > generator; // 产生式表

map<char, set<char> > firstset; // first集合

map<char, set<char> > followset; // follow集合

vector<bool> buildedfirst; // 记录该符号是否已经构造过first集合

vector<bool> buildedfollow; // 记录该符号是否已经构造过follow集合

char start; // 文法的起始符号

map<char, vector<char>> son; // 记录follow集合的依赖关系

vector<bool> havepassfollow; // 记录此依赖关系是否已经传递过follow集合，避免重复传递

vector<map<char, vector<Content>>> states;

// notend存储所有状态；每个状态有多个左部符号；每个左部符号映射至多个产生式右侧

map<int, map<char, int>> jumptable; // 状态-跳转符号-下一状态

map<int, map<char, int>> Rtable; // 归约表 状态-跳转符号-产生式编号

map<int, pair<char, vector<char>>> idtogen; // 将产生式id转换为generator

map<pair<char, vector<char>>, int> gentoid; // 将generator转换为id

vector<int> statestack;

vector<char> sigstack;

vector<char> inputstack;

map<pair<char, int>, vector<pair<char, int>>> fa; // 在构建状态时建立的forward集合之间的依赖关系 当左侧更新时也要更新右侧

// 左侧符号-content的编号

};

(1) 文法的起始符

char start;

(2) 文法终结符&非终结符

vector<char> notend;

vector<char> endsig;

(3) 产生式表

map<char, vector<vector<char>>> generator;

<1> 第一个char代表产生式左部符号；

<2> 第一层vector表示产生式右部符号串的集合；

<3> 最内层vector表示一个符号串；

(4) first集合&follow集合

map<char, set<char>> firset;

map<char, set<char>> followset;

(5) 集合之间的依赖关系

<1> follow集合之间的依赖关系

map<char, vector<char>> son;

需要将所有孩子(vector<char>中的每一个非终结符)的follow集合插入到父亲的follow集合中(第一个char)

<2> forward集合之间的依赖关系

map<pair<char, int>, vector<pair<char, int>>> fa;

在构建项目集是建立的forward结合之间的依赖关系，当左侧更新时(插入新的forward符号)，右侧也要随之更新；

(6) 标记

vector<bool> buildedfirst; // 标记每个非终结符，是否已经构造过first集合

vector<bool> buildedfollow; // 标记每个非终结符，是否已经构造过follow集合

vector<bool> havepassfollow; // 标记follow集合的依赖关系，是否已经通过这个集合传递过

(7) 项目集

vector<map<char, vector<Content>>> states;

<1> 最外层的vector存储所有项目集;

<2> 每一个项目集的主键存储其中每个产生式的左部符号, vector<Content>存储以左部符号为主键的所有Content;

<3> 每一个Content中都存储了vector<char> readed, vector<char> waitread和vector<char> forward

(8) 分析表

map<int, map<char, int>> jumptable; // goto表，其中第一个int表示当前状态，char表示当前遇到的跳转符号，第二个int表示跳转到的状态；

map<int, map<char, int>> Rtable; // 归约表，其中第一个int表示当前状态，char表示当前遇到的跳转符号，第二个int表示归约所用产生式编号；

(9) 产生式id与产生式转换表

map<int, pair<char, vector<char>>> idtogen; // 将产生式id转换为产生式

map<pair<char, vector<char>>, int> gentoid; // 将产生式转换为产生式id

(10) 分析栈

vector<int> stated;

vector<char> sigstack;

(11) 输入符号栈

vector<char> inputstack;

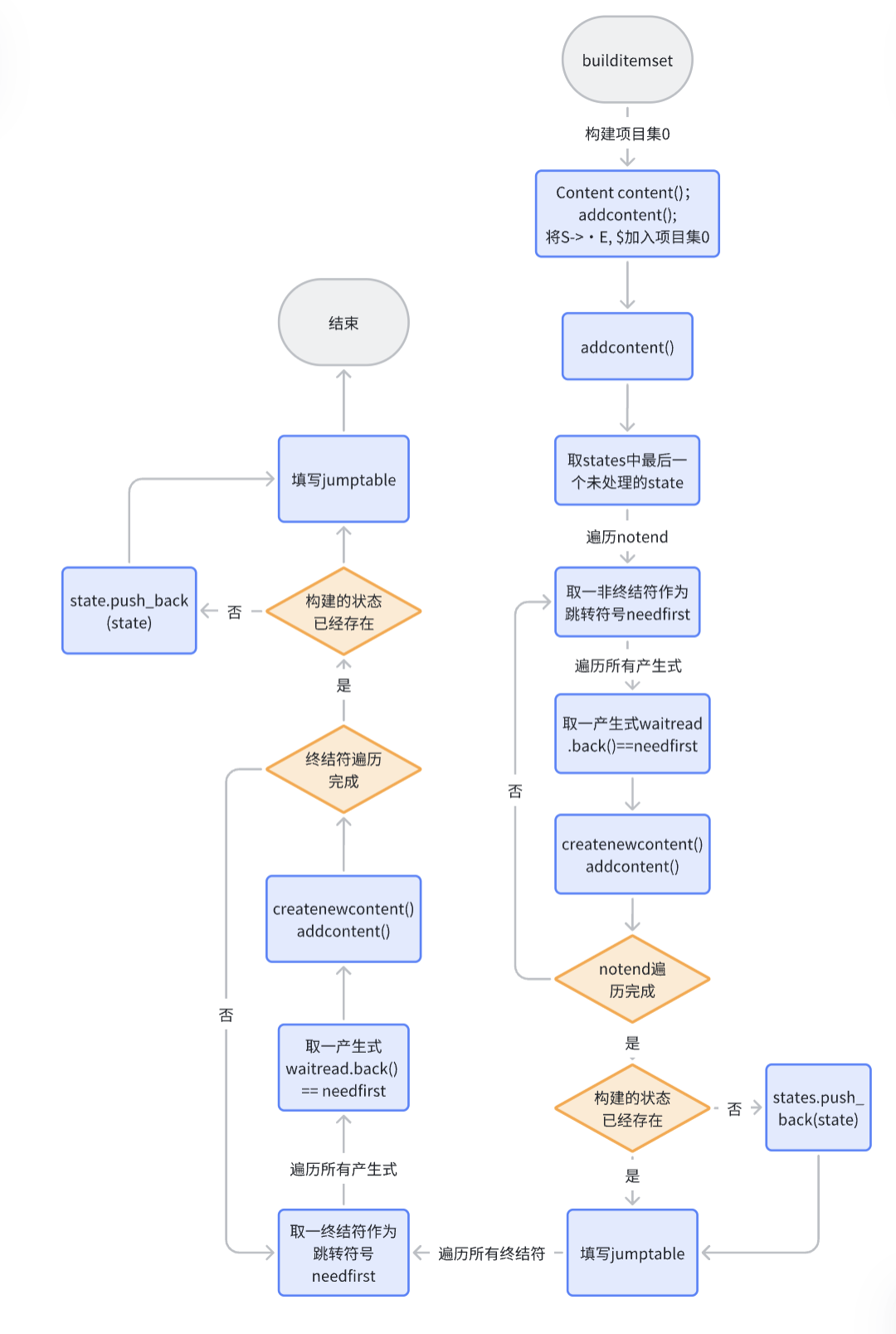
#### 2. 详细设计

##### 2.1 first集合与follow集合的构造

同LL(1)程序

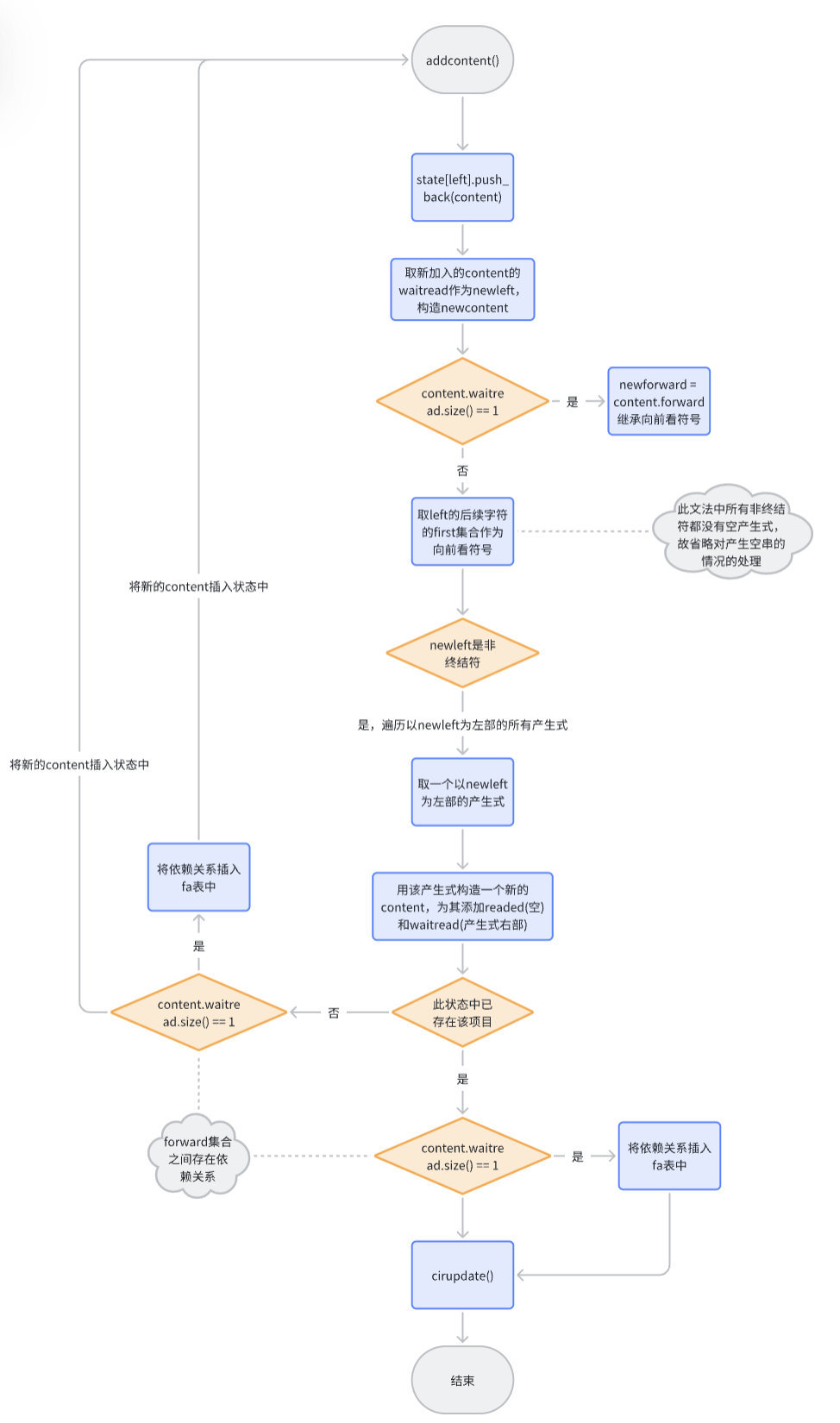
##### 2.2 LR(1)项目集与jumptable的构造

(1) 总体流程



(2) 其中的addcontent()函数

将content及其衍生的所有content插入状态中



(3) 其中的cirupdate函数

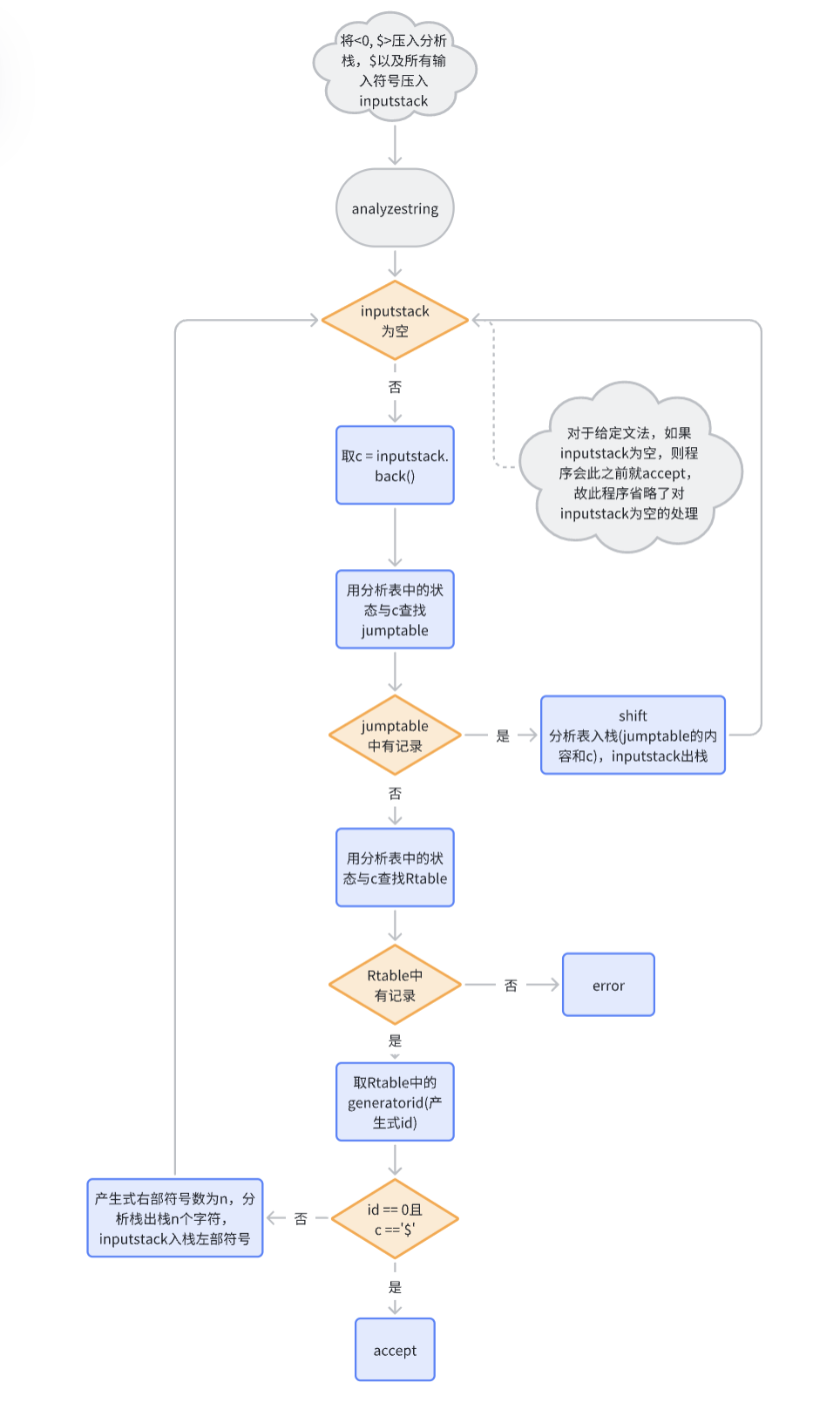
为对应state中的content插入新的forward符号；

遍历依赖这个content的其他content，插入forward符号；

(4) 其中的createnewcontent函数

将原先的content.readed入栈(content.waitread.back())，content.waitread出栈，lookforward不变，构造新的content；

##### 2.4 预测分析程序

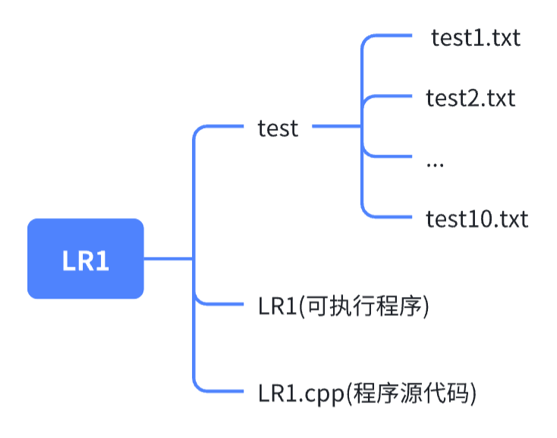


##### 2.5 <<运算符重载

输出start, endsig(终结符), notend(非终结符), Generator(产生式), Firstset(first集合), Followset(follow集合), Itemset(所有项目集), Jumptable(跳转表), Rtable(递归表)信息

### 源程序

#### 1. 提交文件结构如下



#### 2. 编译指令

g++ LR1.cpp -o LR1 -std=c++11

#### 3. 中间信息输出

将main函数中cout << table一行的注释取消，即可输出过程信息

### （四）可执行程序

1. LL1/LL1

2. 执行方式

1. ./LR1 <test/test1.txt
2. ./LR1 程序运行后从标准输入输入一行待分析的字符串

### （五）测试报告

#### 1. 头歌平台测试

#### 2. 本地测试

##### 2.1 ./LR1 <test/test1.txt >output1.txt

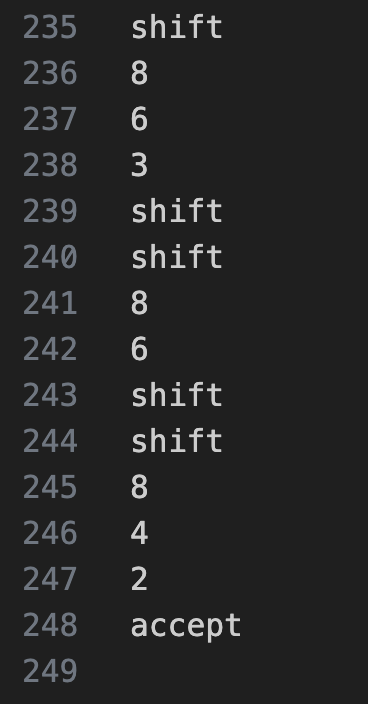
(1) 答案输出如下

##### 

(2) 详细输出见output1.txt文件

##### 2.2 ./LR1 <test/test2.txt >output2.txt

(1) 答案输出如下



(2) 详细输出见output2.txt文件