# 《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**

# 一、LL(1)语法分析程序

## （一）实验题目

### 1. 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

| 编号 | 产生式 |
| --- | --- |
| 1 | E -> TA |
| 2 | A -> +TA |
| 3 | A -> -TA |
| 4 | A -> ε |
| 5 | T -> FB |
| 6 | B -> *FB |
| 7 | B -> /FB |
| 8 | B -> ε |
| 9 | F -> (E) |
| 10 | F -> num |

（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；

### 2. 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

### 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：一个算术表达式，构成该算术表达式的字符有：{ 'n', '+', '-', '*', '/', '(', ')' }

(2) 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：

a. 分析栈：以$符号表示栈底的字符串(左侧表示栈底，由终结符和非终结符构成);
b. 输入串栈：以$符号表示栈底的字符串(右侧为栈底);
c. 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)。

(3) 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 |

# （二）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 的构造函数

a) 构建 start，文法的起始符；
b) 构建 notends，文法的非终结符；
c) 构建 characters，文法的终结符，用 n 代表 num，e 代表空；
d) 将所有产生式存入 table 中，作为 generator；
e) 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)；
f) 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)；
g) 初始化 reachempty 集合，将所有能产生空串的字符存入 set 中；
h) 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；
i) 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；

(3) buildfollowset()

a) 将$符加入起始符的 follow 集合；

b) 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；

c) 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；

d) 去掉 follow 集合中的 e，即空产生式；

(4) buildtable()

a) 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；
};
```

(5) analyzeinput()

a) 将标准输入压入输入符号栈；

b) 从分析栈中取出 left，从符号栈中取出 c,

i. 若 left 与 c 相等，则 match；

ii. 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；

## 1.2 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

(2) 终结符表&非终结符表

a) vector<char> characters; 终结符表

b) vector<char> notends; 非终结符表

(3) 产生式表

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

b) 主键代表产生式左部符号;

c) vector<vector<char>> 代表产生式左部符号相同的所有产生式右部;

d) 最里层的 vector<char>存储了一个产生式右部的所有符号;

(4) first 集&follow 集合

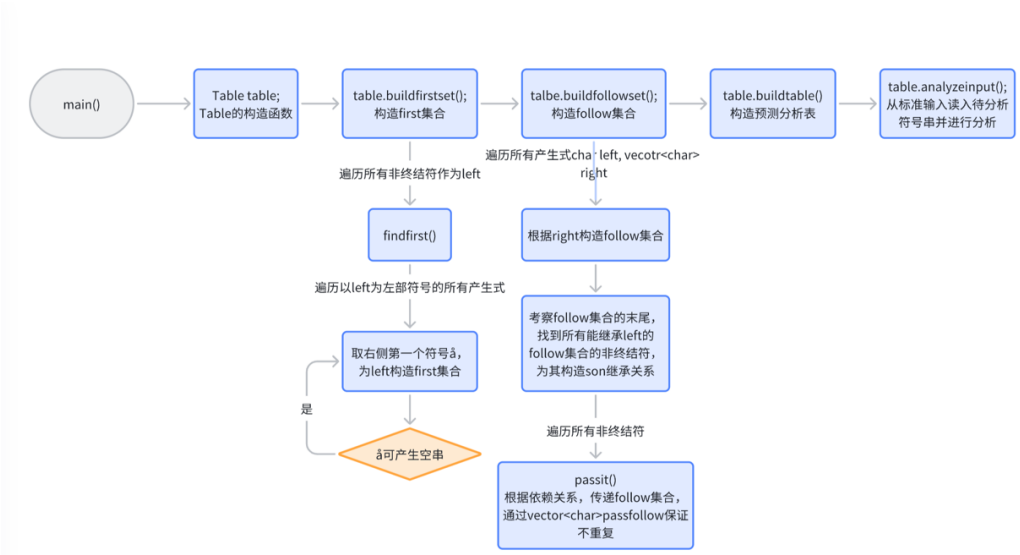
a) map<char, set<char>> firstset；主键代表一个非终结符，set<char>表示非终结符对应的 first 集合；
b) map<char, set<char>> followset； 主键代表一个非终结符，set<char>表示非终结符对应的 follow 集合；

(5) 分析表
a) map<char, map<char, vector<char>>> M;
b) 第一层主键代表非终结符；
c) 第二层主键代表终结符；
d) 最内层代表需要使用的产生式的右部符号；
e) 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)；

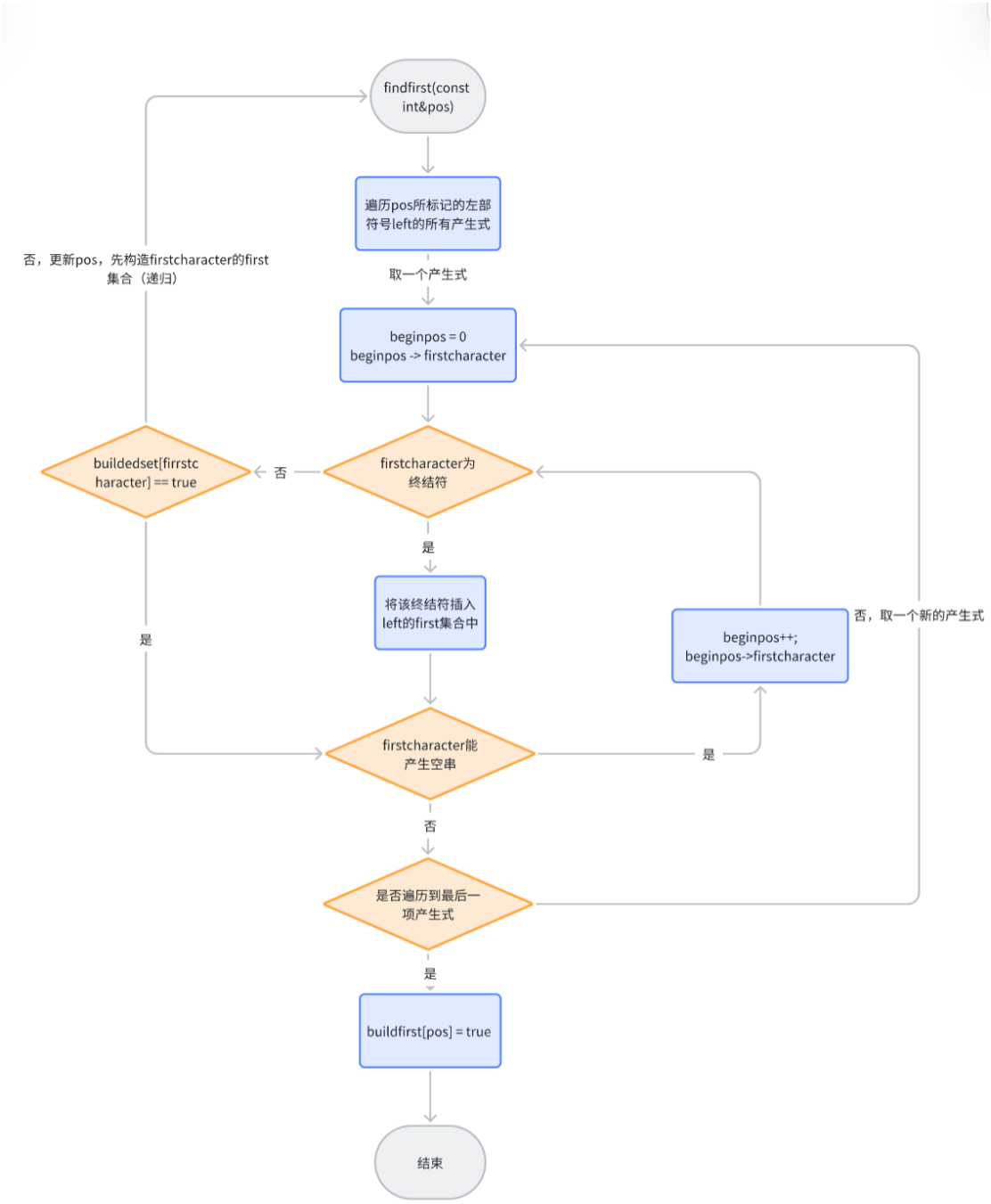
(6) 分析栈&符号栈
a) vector<char> analyzestack;
b) vector<char> inputcharstack;
c) push_back()入栈，pop_back()出栈

(7) id 表
a) map<char, map<char, int>>id;
b) 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；

## 2. 详细设计

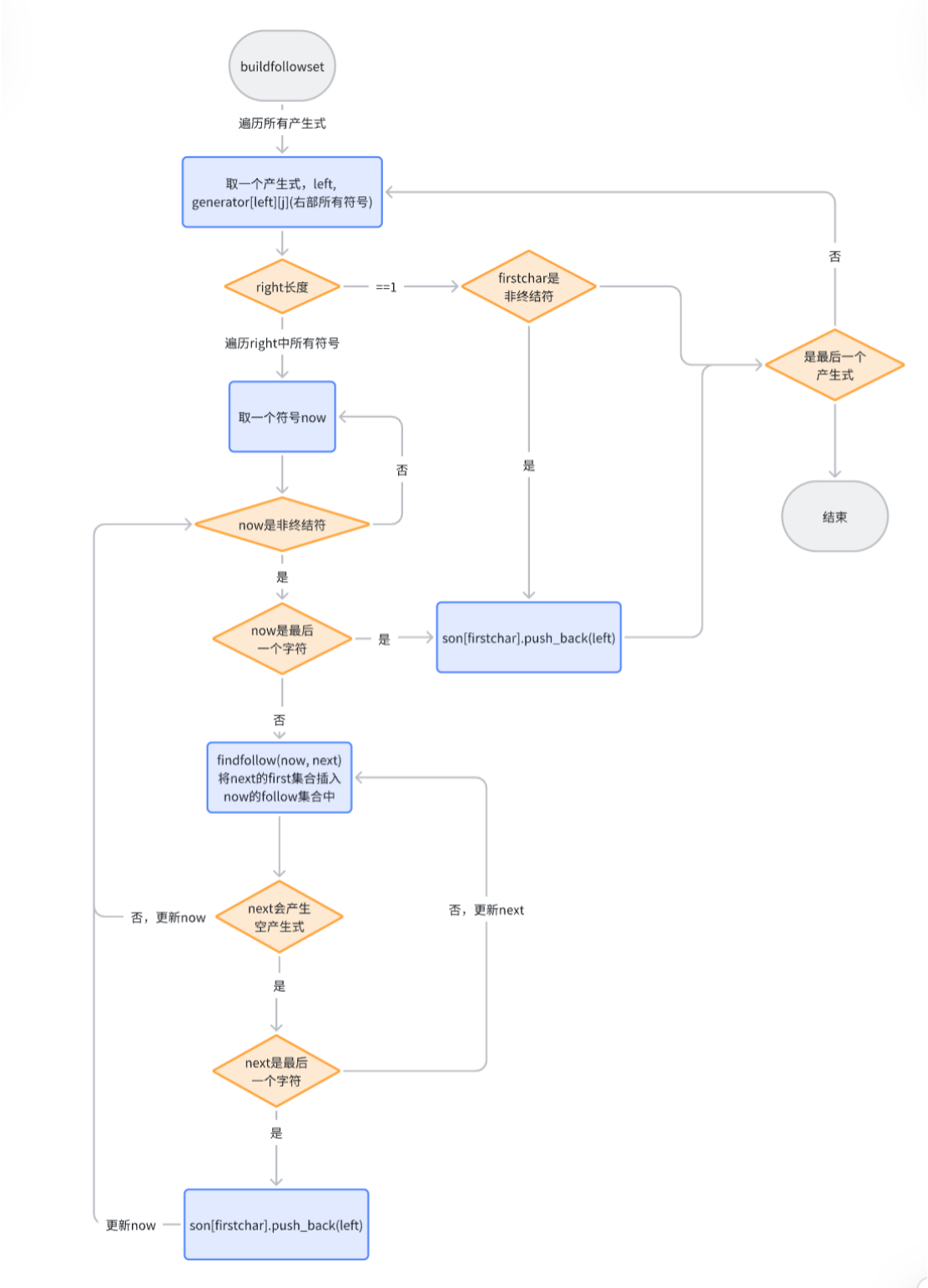


## 2.1 first 集合的构造

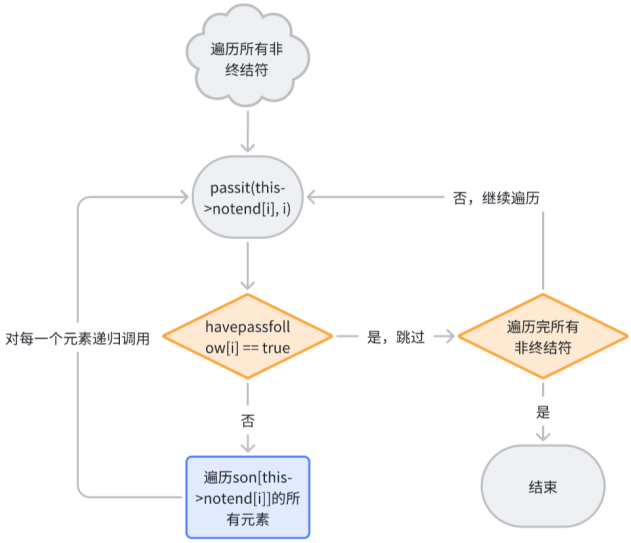



## 2.2 follow 集合的构造

(1) 第一步直接构造

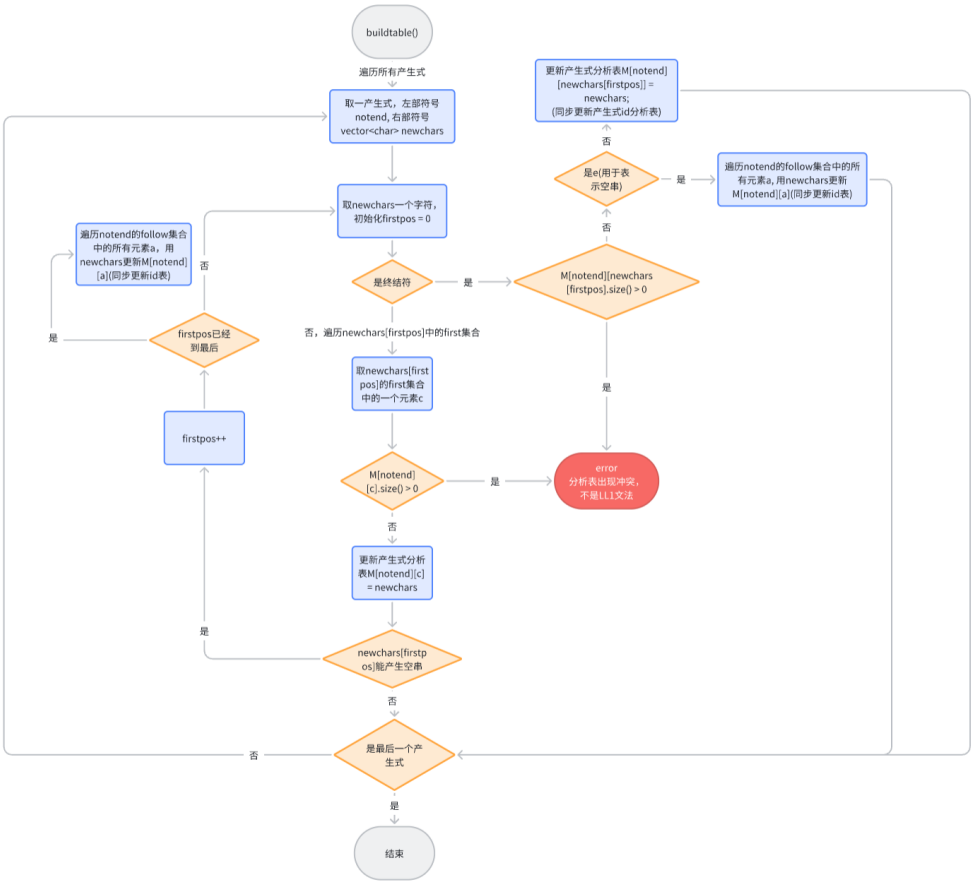

buildfollowset
遍历所有产生式
取一个产生式，left,
generator[left][j](右部所有符号)
right长度
==1
firstchar是
非终结符
否
是最后一个
产生式
结束
遍历right中所有符号
取一个符号now
否
是
now是非终结符
是
now是最后
一个字符
是
son[firstchar].push_back(left)
否
findfollow(now, next)
将next的first集合插入
now的follow集合中
next会产生
空产生式
否，更新now
否，更新next
是
next是最后
一个字符
是
更新now
son[firstchar].push_back(left)

（2）follow 集合的传递；

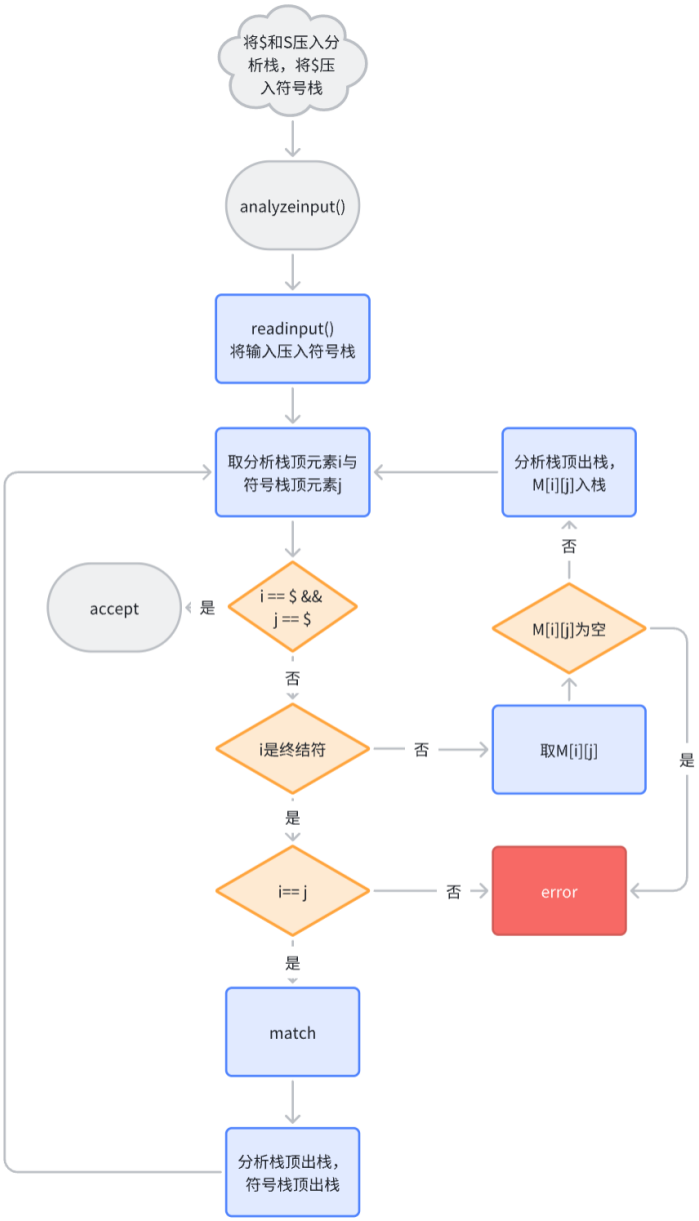



（3）去掉所有 follow 集合中的空串；

## 2.3 分析表的构造



## 2.4 分析程序

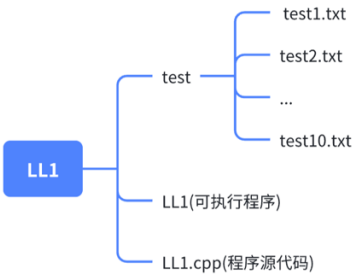



## 2.5 过程输出(<<运算符重载)

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

# （三）源程序

## 3.1 提交文件结构如下：




## 3.2 编译指令

a) 在 LL1/下

b) 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；

# （四）可执行程序

1. LL1/LL1
2. 执行方式

(1) ./LL1 <test/test1.txt

(2) ./LL1 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

## 5.1 头歌平台测试


## 5.2 本地测试

### 1. test1.txt

```
(base) mac@macdeMacBook-Pro LL1 % g++ LL1.cpp -o LL1 -std=c++11
(base) mac@macdeMacBook-Pro LL1 % ./LL1 <test/test1.txt
$E      n+n$    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
$       $       accept
start:
E

Characters:
+ - * / ( ) n e $

notends:
E T A F B

Generator:

E -> TA

T -> FB

A -> +TA
A -> -TA
A -> e

F -> (E)
F -> n

B -> *FB
B -> /FB
B -> e

Firstset:
E: { ( n }
T: { ( n }
Followset:
E: { $ ) }
T: { $ ) + - }
A: { $ ) }
F: { $ ) * + - / }
B: { $ ) + - }

Table:
        +       -       *       /       (       )       n       $
E                                       TA              TA
T                                       FB              FB
A       +TA     -TA                             e               e
F                                       (E)             n
B       e       e       *FB     /FB             e               e

Id:
0       +       -       *       /       (       )       n       $
E       0       0       0       0       1       0       1       0
T       0       0       0       0       5       0       5       0
A       2       3       0       0       0       4       0       4
F       0       0       0       0       9       0       10      0
B       8       8       6       7       0       8       0       8
```

2. test2.txt

```
(base) mac@macdeMacBook-Pro LL1 % ./LL1 <test/test2.txt
$E      n-n*n$  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
start:
E

Characters:
+ - * / ( ) n e $

notends:
E T A F B

Generator:

E -> TA

T -> FB

A -> +TA
A -> -TA
A -> e

F -> (E)
F -> n

B -> *FB
B -> /FB
B -> e

Firstset:
E: { ( n }
T: { ( n }
A: { + - e }
F: { ( n }
B: { * / e }

Followset:
E: { $ ) }
T: { $ ) + - }
A: { $ ) }
F: { $ ) * + - / }
B: { $ ) + - }

Table:
        +       -       *       /       (       )       n       $
E                                       TA              TA
T                                       FB              FB
A       +TA     -TA                             e               e
F                                       (E)             n
B       e       e       *FB     /FB             e               e

Id:
0       +       -       *       /       (       )       n       $
E       0       0       0       0       1       0       1       0
T       0       0       0       0       5       0       5       0
A       2       3       0       0       0       4       0       4
F       0       0       0       0       9       0       10      0
B       8       8       6       7       0       8       0       8
```

# 二、LR(1)语法分析程序

## （一） 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：

| 编号 | 产生式 |
| --- | --- |
| 0 | E' -> E |
| 1 | E -> E+T |
| 2 | E -> E-T |
| 3 | E -> T |
| 4 | T -> T*F |
| 5 | T -> T/F |
| 6 | T -> F |
| 7 | F -> (E) |
| 8 | F -> num |

（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

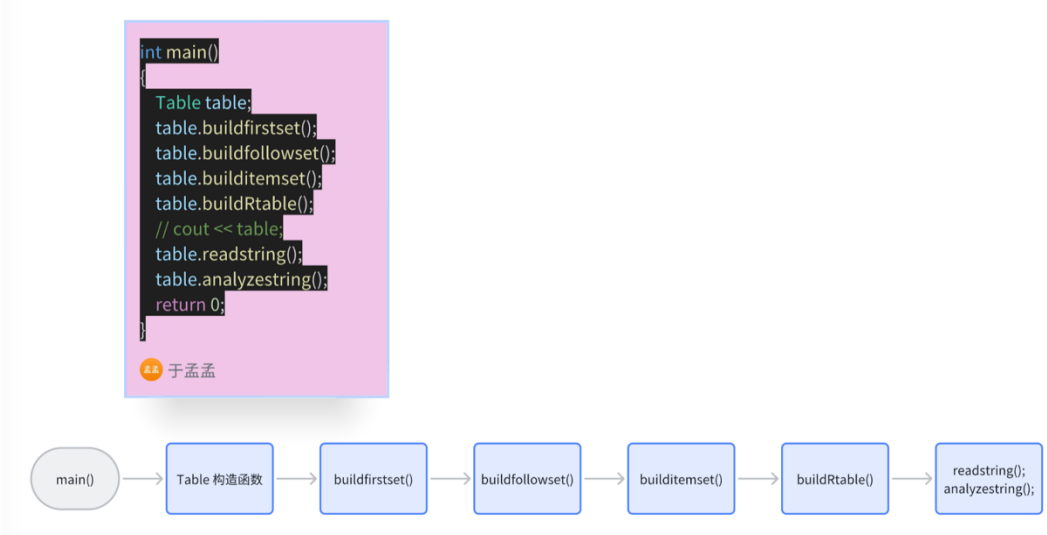
1

accept

# （二） 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；

**(1) I中的每一个项目都属于closure(I)；**
**(2) 若项目[A→α•Bβ, a]属于closure(I)，且**
**G有产生式B→η，则**
**对任何终结符号 b∈FIRST(βa)，**
**若项目[B→•η, b]不属于集合closure(I)，**
**则将它加入closure(I)；**
**(3) 重复规则(2)，直到closure(I)不再增大为止。**

<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;

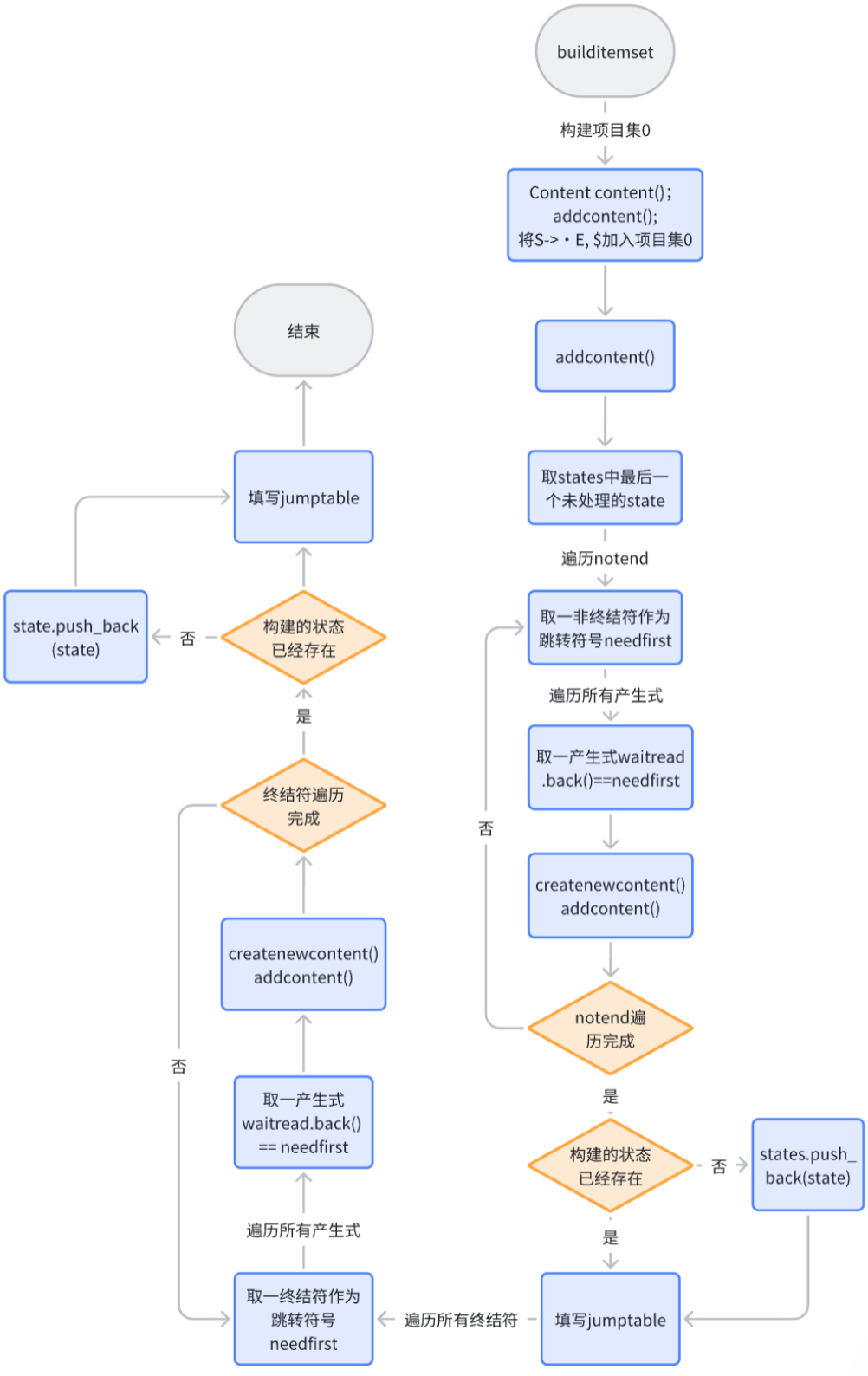
# 2. 详细设计

## 2.1 first 集合与 follow 集合的构造

同 LL(1)程序

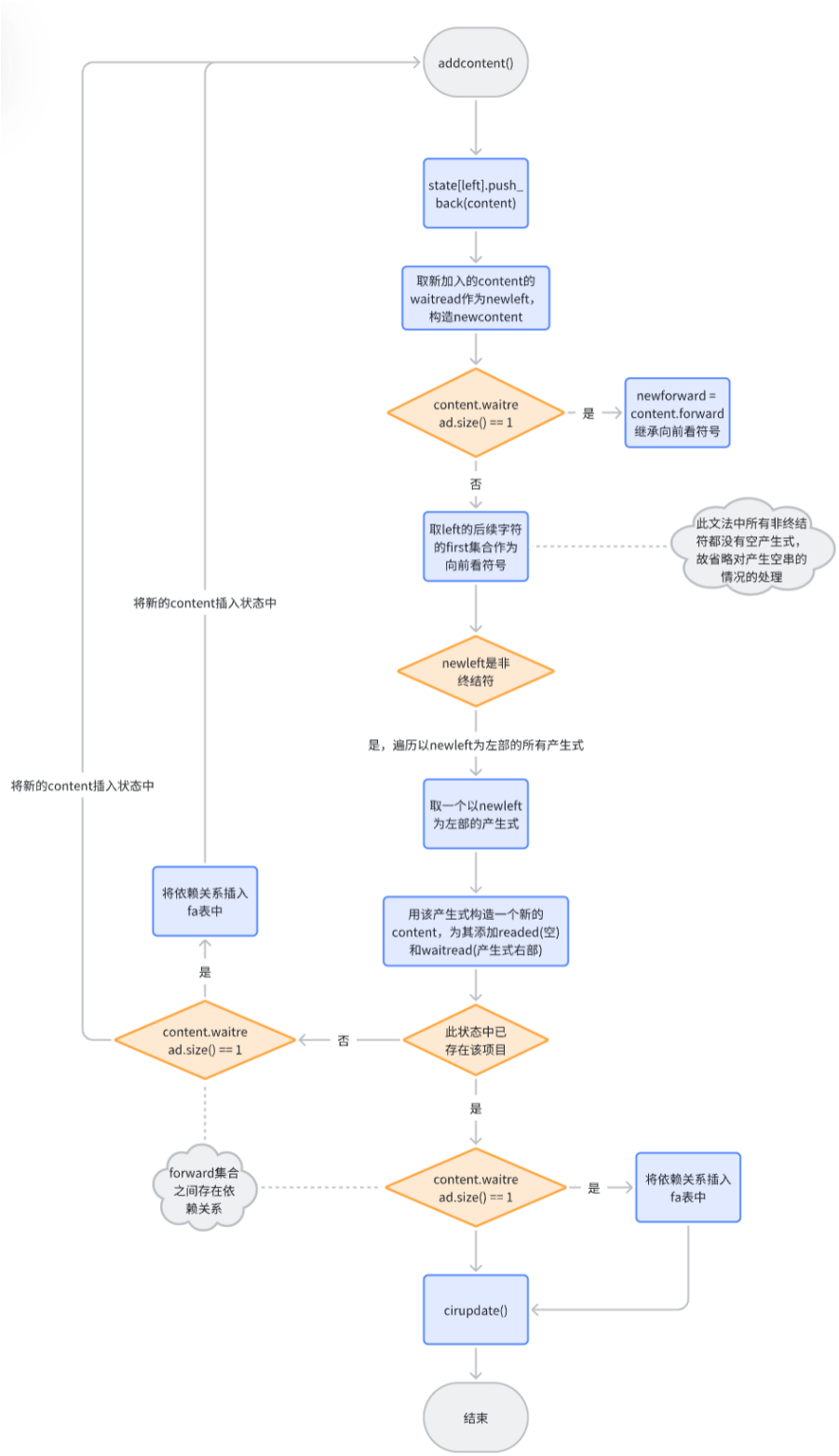
## 2.2 LR(1)项目集与 jumptable 的构造

(1) 总体流程


builditemset
构建项目集0
Content content();
addcontent();
将S->·E, $加入项目集0
addcontent()
取states中最后一个未处理的state
遍历notend
取一非终结符作为跳转符号needfirst
遍历所有产生式
取一产生式waitread.back()==needfirst
createnewcontent()
addcontent()
notend遍历完成
否
是
构建的状态已经存在
否
states.push_back(state)
是
填写jumptable
遍历所有终结符
取一终结符作为跳转符号needfirst
遍历所有产生式
取一产生式waitread.back() == needfirst
createnewcontent()
addcontent()
否
终结符遍历完成
是
构建的状态已经存在
否
state.push_back(state)
填写jumptable
结束

(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

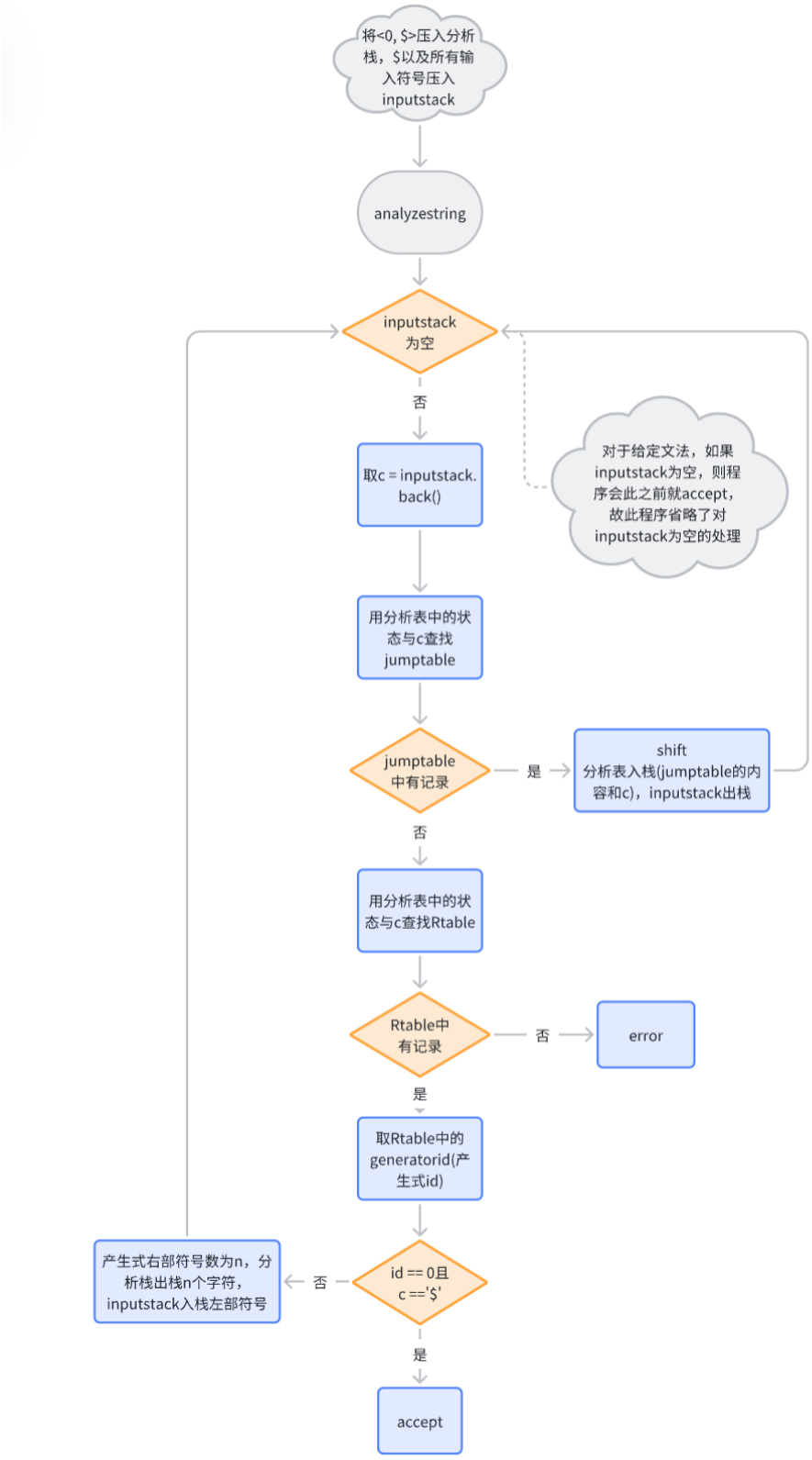
为对应 state 中的 content 插入新的 forward 符号；

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

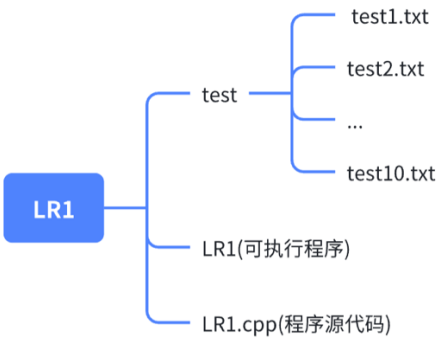

将<0, $>压入分析栈，$以及所有输入符号压入inputstack
analyzestring
inputstack为空
否
取c = inputstack.back()
对于给定文法，如果inputstack为空，则程序会此之前就accept，故此程序省略了对inputstack为空的处理
用分析表中的状态与c查找jumptable
jumptable中有记录
是
shift
分析表入栈(jumptable的内容和c)，inputstack出栈
否
用分析表中的状态与c查找Rtable
Rtable中有记录
否
error
是
取Rtable中的generatorid(产生式id)
id == 0且c =='$'
否
产生式右部符号数为n，分析栈出栈n个字符，inputstack入栈左部符号
是
accept

### 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

## （三） 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

(3) ./LR1 <test/test1.txt
(4) ./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

```
235  shift
236  8
237  6
238  3
239  shift
240  shift
241  8
242  6
243  1
244  accept
```

(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

```
235   shift
236   8
237   6
238   3
239   shift
240   shift
241   8
242   6
243   shift
244   shift
245   8
246   4
247   2
248   accept
249
```

(2) 详细输出见 output2.txt 文件