



本文档源链接: https://axt9hz09lxr.feishu.cn/wiki/BfDrwX0xWiy0jvkNHHTcoWDknjh?edition_id=k8C7jH

编译原理 复习提纲（2021级必拿下编译原理👊）

🕒 2023-2024 第一学期

😊 本文档主要贡献者:

计算机学院 2021级:

李达良 翁行 邹锐城 郑伟楠 曾嘉杰 李倩旎 关竣佑 陈俊豪 郑方形 刘喜行 洪福林 彭炜钦

人工智能学院 2021级: 余昕芪

ps: 可能还有一些没列出来的贡献者, 但由于修改记录过多, 可能没注意到, 在此一并感谢

! 阅前须知:

1. 由于时间紧迫, 本文档可能仍然含有一定错误和缺漏, 请自行补充、更正。
2. 本文档对于四元组等课程较后面的知识比较欠缺, 请自行补充完善。
3. 请紧紧围绕当年hyl提供的课件以及实验对本文档进行补充或者更正, 以达到最佳的复习效果。

📌 2023-2024学年度第一学期 计算机学院 期末试题:

- 一、第一题是自顶向下文法需要满足什么条件, 不满足要怎么做
- 二、C语言十进制数的正则和DFA图
- 三、手写slr1判断代码
- 四、自顶向下写出正则表达式的后缀表达式生成
- 五、写出TINY程序if else语句的四元组生成属性文法
- 六、改写TINY, 写出C语言的do while文法, 写出词法和语法分析程序

第一题：在自顶向下分析中，文法需要满足什么条件，如果不满足如何修改（消除左递归和左公因子和二义性）


第二题：十进制数的正则表达式和画出dfa（最抽象和最简单的一次，居然没有考代码）

答案如下，课上讲过的原题

带科学计算的浮点数：

```
1 nat = [0-9]+
2 signedNat = (+|-)?nat
3 number = signedNat("."nat)?(E signedNat)?
```

DFA:

 2024.1编译原理回忆版(1).pdf


2023-2024AI编译原理考题：（2023/1/3日，人工智能学院编译原理试题，实验与我们完全一致，老师也是只有浴帘）

2023-2024《编译原理》回忆版

AI限定

题型及占分比例

- 一、基础知识点（1题，15分）
- 二、正则表达式→DFA分析题（1题，15分）

 4_2023-2024《编译原理》回忆版.pdf

 目录导航

一、基础知识点

题型一：正则表达式与文法互转

参考链接：

[编译原理-正则文法与正则表达式的相互转化_正则表达式转换-CSDN博客](#)

[编译原理——正规式转换为正规文法_为正规文法g\[s\] s→aa|bq a→aa|bb|b b→bd|aq q→aq|bdb d→bb|-CSDN博客](#)

例题：

1. 我们知道：正则文法与正则表达式之间是可以相互转换的，请分别完成以下的转换。(1) 将正则表达式 $(0|1)^*11(0|1)^*$ 转换为正则文法。(2) 将正则文法 $G[S]$ 转换为正则表达式。 $G[S]=\{ \quad S \rightarrow 0B \quad B \rightarrow 0B|1S|0 \quad \}$

参考答案：（注意是正则文法，即3型文法）

(1)

$A \rightarrow r$ 是正则定义式，则对 $A \rightarrow r$ 的分解规则如下：

(1) 如果 $r=r_1r_2$ ，则将 $A \rightarrow r$ 分解为 $A \rightarrow r_1B$ ， $B \rightarrow r_2$ ， $B \in V$ ；

(2) 如果 $r=r_1^*r_2$ ，则将 $A \rightarrow r$ 分解为 $A \rightarrow r_1A$ ， $A \rightarrow r_2$ ；

(3) 如果 $r=r_1|r_2$ ，则将 $A \rightarrow r$ 分解为 $A \rightarrow r_1$ ， $A \rightarrow r_2$ 。

不断应用分解规则(1)到(3)对各个正则定义式进行分解，直到每个正则定义式右端只含一个语法变量(即符合正则文法产生式的形式)为止。

http://blog.csdn.net/weixin_458243

```
1 S -> (0|1)*11(0|1)*
2 -----
3 S -> 0S | 1S | 11(0|1)*
4 -----
```

```
5 S -> 0S | 1S | 1A
6 A -> 1(0|1)*
7 -----
8 S -> 0S | 1S | 1A
9 A -> 1B
10 B -> (0|1)*ε
11 -----
12 最终答案：
13 S -> 0S | 1S | 1A
14 A -> 1B
15 B -> 0B | 1B | ε
```

(2) 逆向

```
1 S -> 0B
2 B -> 0B | 1S | 0
```

```
3 -----
4 B -> 0B | 10B | 0
5 -----
6 B -> (0|10)B | 0
7 -----
8 B -> (0|10)*0
9 -----
10 最终答案：（代入S->0B）
11 0(0|10)*0
```

题型二：写出某个C++语法的正则表达式，画出DFA和词法分析代码段（复习课着重）

常见的正则：

1. 数

整数：

```
1 nat = [0-9]+
2 signedNat = (+|-)?nat
```

八进制：0[0-7]+

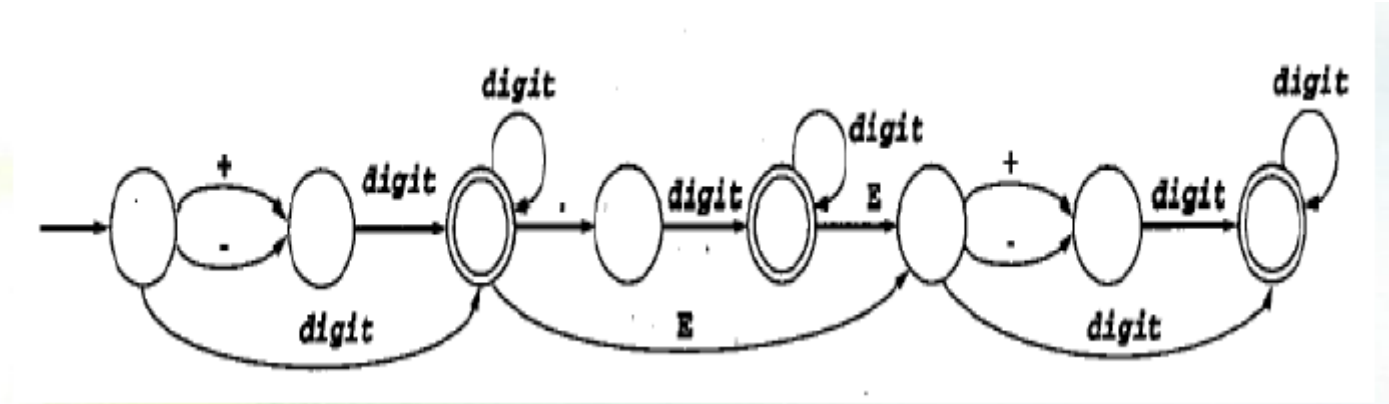
十六进制：0[xX][0-9a-fA-F]+

---上面的比较简单，不做分析，其实下面那个摘下来一部分就是---

带科学计算的浮点数：

```
1 nat = [0-9]+
2 signedNat = (+|-)?nat
3 number = signedNat("."nat)?(E signedNat)?
```

DFA:



词法分析程序：

```
1 #include<iostream>
2 #include<string>
3 using namespace std;
4 int main()
5 {
6     string input;
7     cin >> input;
8     int currentState = 1;
9     int length = input.length();
10    for (int i = 0; i < length; i++)
11    {
```

```

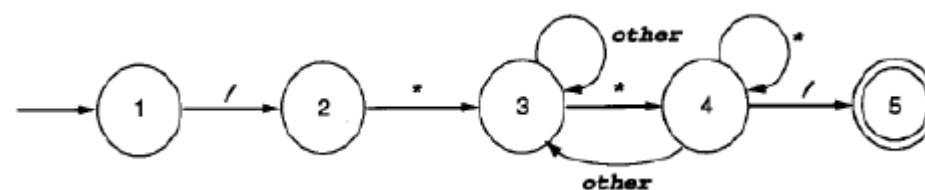
12     char c = input[i];
13     switch (currentState)
14     {
15     case 1:
16         switch (c)
17         {
18         case '+':
19         case '-':
20             currentState = 2;
21             break;
22         case '1':
23             //... (数字)
24         case '9':
25             currentState = 3;
26             break;
27         default:
28             cout << "error" << endl;
29         }
30         break;
31         //... (将所有状态照葫芦画瓢弄出来)
32     }
33     // 终态的switch
34     switch (currentState) {
35     case 3:
36     case 5:
37     case 8:
38         cout << "Accepted" << endl;
39         break;
40     default:
41         cout << "Not Accepted" << endl;
42     }
43     return 0;
44 }
45 }

```

2. 多行注释

这个没有正则表达式，比较困难

DFA:



词法分析程序：

(技巧：结构是差不多的，初始的int currentState = 1; 然后，遇到什么符号，switch case就变换currentState到状态几，最后再来个 switch (currentState)，终态就ACCEPT，不然就NOT ACCEPTED)

```

1  #include<iostream>
2  #include <string>
3  using namespace std;
4  int main() {
5      string input;
6      cout << "Enter input string: ";
7      cin >> input;
8      int currentState = 1;
9      int length = input.length();
10     for (int i = 0; i < length; i++) {

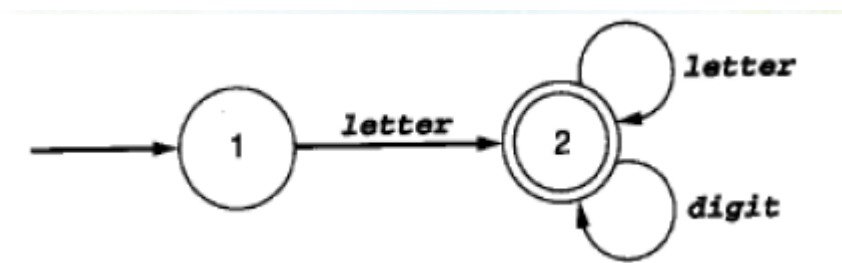
```

```
11     char c = input[i];
12     switch (currentState) {
13     case 1:
14         switch (c)
15         {
16             case '/':
17                 currentState = 2;
18                 break;
19             default:
20                 cout << "Error" << endl;
21                 return 1;
22         }
23         break;
24     case 2:
25         switch (c)
26         {
27             case '*':
28                 currentState = 3;
29                 break;
30             default:
31                 cout << "Error" << endl;
32                 return 1;
33         }
34         break;
35     case 3:
36         switch (c)
37         {
38             case '*':
39                 currentState = 4;
40                 break;
41             default:
42                 currentState = 3;
43         }
44         break;
45     case 4:
46         switch (c)
47         {
48             case '*':
49                 currentState = 4;
50                 break;
51             case '/':
52                 currentState = 5;
53                 break;
54             default:
55                 currentState = 3;
56         }
57         break;
58     case 5:
59     default:
60         cout << "Error" << endl;
61         break;
62     }
63 }
64 switch (currentState) {
65 case 5:
66     cout << "Accepted" << endl;
67     break;
68 default:
69     cout << "Not Accepted" << endl;
70 }
71 return 0;
72 }
```

3. 标识符

正则: $l(l|d)^*$

DFA:



词法分析程序:

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 int main() {
7     string input;
8     cout << "Enter input string: ";
9     cin >> input;
10    int currentState = 1;
11    int length = input.length();
12    for (int i = 0; i < length; i++) {
13        char c = input[i];
14        switch (currentState) {
15            case 1:
16                switch (c) {
17                    case 'd':
18                        cout << "Error: Invalid input character '" << c << "'" << endl;
19                        return 1;
20                        break;
21                    case 'l':
22                        currentState = 2;
23                        break;
24                    default:
25                        cout << "Error: Invalid input character '" << c << "'" << endl;
26                        return 1;
27                }
28                break;
29            case 2:
30                switch (c) {
31                    case 'd':
32                        currentState = 2;
33                        break;
34                    case 'l':
35                        currentState = 2;
36                        break;
37                    default:
38                        cout << "Error: Invalid input character '" << c << "'" << endl;
39                        return 1;
40                }
41                break;
42        }
43    }
44    switch (currentState) {
45        case 2:
```

```

45         cout << "Accepted" << endl;
46         break;
47     default:
48         cout << "Not Accepted" << endl;
49     }
50     return 0;
51 }
52

```

题型三：给一个实际例子，写出正则表达式、DFA图

1. b后面一定跟着a

正则： $(a|ba)^*$

DFA:

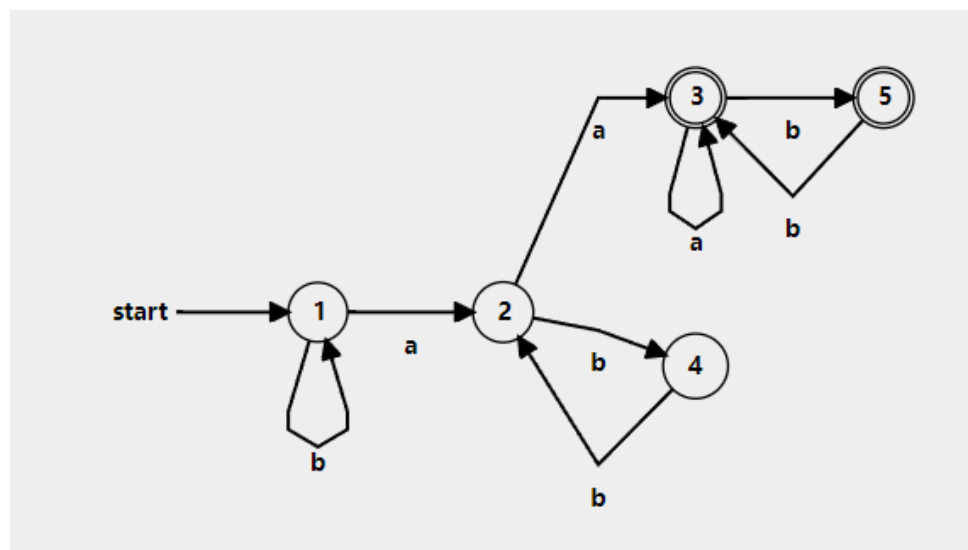


(如果转成文法怎么转，参考题型一)

2. 至少两个a，且任何两个a之间有偶数个b

正则： $b^*a(bb)^*a(a|(bb)^*)^*b^*$

DFA:



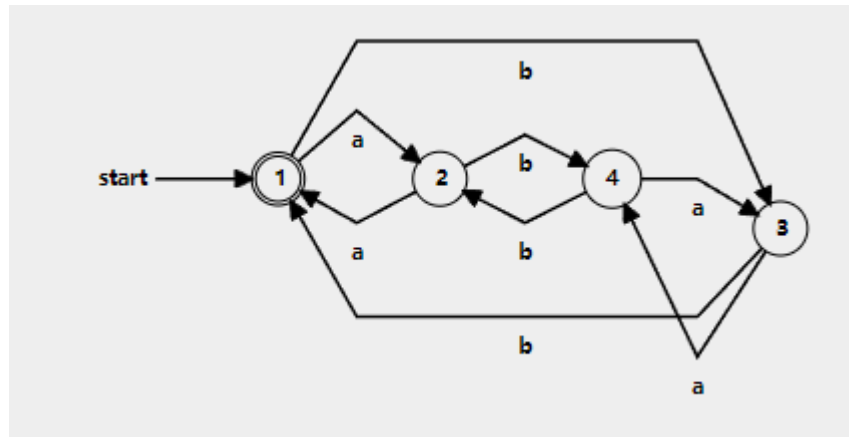
文法参考（题型一）：

- 1 $S \rightarrow bS \mid aB$
- 2 $B \rightarrow bC \mid aD$
- 3
- 4 $C \rightarrow bB$
- 5 $D \rightarrow aD \mid bE \mid bD \mid \epsilon$
- 6 $E \rightarrow bD$

3. 偶数个a和偶数个b组合在一起

正则: `(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*`

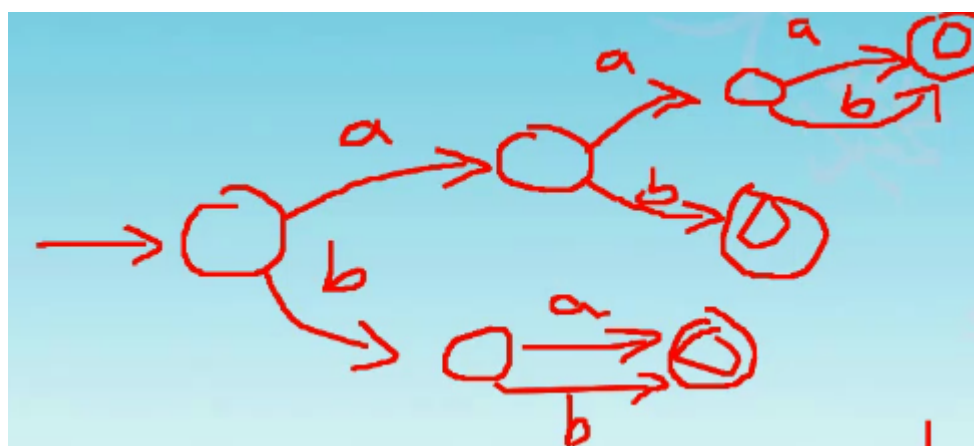
DFA:



4. 自动售票机问题

接受1元和2元纸币，车票要3元，超出3元也只吐出一张车票

先画DFA:



再写正则: `a(a(a|b)|b) | b(a|b)`

语法分析程序

```
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  int main() {
6      string input;
7      cin >> input;
8      int State = 1;
9
10     for (int i = 0; i < input.length(); i++) {
11         char c = input[i];
12         switch (State) {
13             case 1:
14                 if (c == '1') State = 2;
15                 else if (c == '2') State = 3;
16                 else goto ERROR;
17                 break;
18             case 2:
19                 if (c == '1') State = 4;
20                 else if (c == '2') State = 5;
21                 else goto ERROR;
22                 break;
23             case 3:
24                 if (c == '1' || c == '2') State = 6;
25                 else goto ERROR;
26                 break;
27             case 4:
```

```

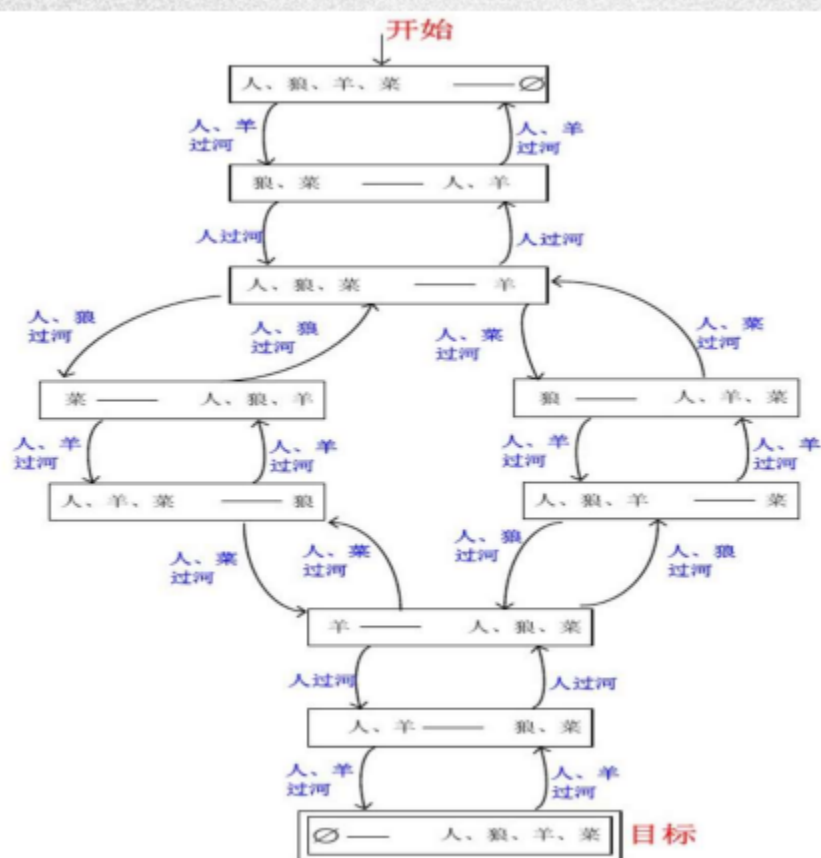
28         if (c == '1' || c == '2') State = 7;
29         else goto ERROR;
30         break;
31     ERROR:
32         cout << "Error input: " << c << endl;
33         return 0;
34     }
35 }
36
37 if (State == 5 || State == 6 || State == 7)
38     cout << "Accept!" << endl;
39 else
40     cout << "Not Accept!" << endl;
41
42 return 0;
43 }

```

5. 狼羊菜问题 (不确定)

DFA:

过河问题



- 从这个图中找出从初始状态到目标状态的一条有向路，就是这个问题一个解。

正则：

(人羊)人 (人狼) (人羊) (人菜) (人菜) (人羊) (人狼) 人 (人羊)

题型四：判断文法是否正确（一般就是优先级错了）

判断该正则表达式文法是否正确？如不正确应该如何进行改写？

$$S \rightarrow RE | RE'' | '' RF | RE \cdot RF | (RE) | RE^* | RE^+$$

答：不正确。没有考虑正则表达式各运算符的优先级。应改写为如下文法：

$$RE \rightarrow RE'' | '' RT | RT$$

$$RT \rightarrow RT \cdot RC | RC$$

$$RC \rightarrow RC^* | RC^+ | RF$$

$$RF \rightarrow (RE) | char$$

其中 *char* 表示单个字符。

其实主要问题是：未考虑正则表达式各运算符的优先级导致了文法二义性的问题（参考PPTchap03的94页）。

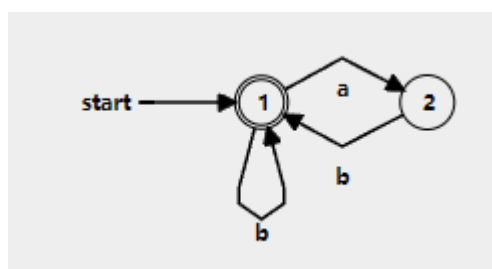
Ps: “” 是双引号，即文法是：RE 或 RE | RF 或 RE*RF 或。。。。

题型五：手撕DFA最小化过程

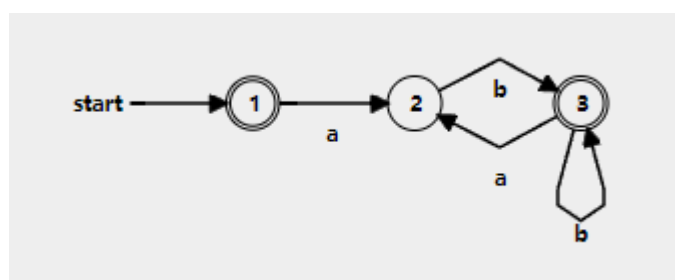
例题：试证明：(ab|b*)*与 (abb*)*等价

！ 试题来自于hyl网课，但该题貌似有点错误，这两个应该是不等价的，理解意思就好。。。

答案：(ab|b*)*最小化结果



(abb*)*最小化结果



题型六：消除文法左递归

1. 改为右递归

例题：消除下面文法的左递归：
 $A \rightarrow Ba | Aa | c$
 $B \rightarrow Bb | Ab | d$

a. 先消除 *A* 的直接左递归：

$$A \rightarrow (Ba | c) A'$$

$$A' \rightarrow aA' | \epsilon$$

b. 将消除了直接左递归的 A 代入后面的文法中

$$B \rightarrow Bb \mid BaA'b \mid cA'b \mid d, \text{ 提取左公因子:}$$

$$B \rightarrow B(b \mid aA'b) \mid cA'b \mid d, \text{ 消除 } B \text{ 的直接左递归}$$

$$B \rightarrow (cA'b \mid d)B'$$
$$B' \rightarrow bB' \mid aA'bB' \mid \epsilon$$

2. 改为EBNF

例题：将下列文法改为EBNF：

```
1 exp -> exp addop term | term
2 addop -> +|-
3 term -> term mulop factor | factor
4 mulop -> *
5 factor -> (exp) | number
```

转化为EBNF的结果：

```
1 exp -> term { addop term }
2 addop -> +|-
3 term -> factor { mulop factor }
4 mulop -> *
5 factor -> (exp) | number
```

题型七：死记硬背题

1. 程序语言的分类

- a. 低级语言（汇编语言）
- b. 高级语言（以类似于自然语言的方式编写）

2. 程序翻译的方式有哪几种？有何不同？

- a. 解释的方法（Python），容易实现跨平台，需要解释器。
- b. 编译的方法（C++），生成目标代码，较难实现跨平台。

Java是介于二者之间的

3. 编译程序包含多少个阶段，各阶段的功能任务分别是什么

- a. 词法分析：识别Token
- b. 语法分析：生成AST
- c. 语义分析：数据类型分析、作用域分析（符号表）
- d. [源代码集优化]（可选）
- e. 中间代码生成（三元组、四元组、逆波兰、树型、伪代码）
- f. 目标代码生成
- g. 根据目标机器的指令特点做出优化（优化运行效率和节省内存空间）

4. 推导、规约、语法树、文法的二义性。

- 推导：左部推导右部
- 规约：右部规约左部
- 文法：非终结符号、终结符号、规则集、开始符号
- 文法分类：

- 0型文法：无限制文法或短语文法
- 1型文法：上下文有关
- 2型文法：上下文无关
- 3型文法：正则文法或正规文法

5. 文法二义性的消除方法有多少种：3+2种

- a. 加入限制规则（自然语言说明）
- b. 改造文法规则
- c. 重新设计书写方法

例如：悬挂else问题，既可以改文法，也可以重新设计书写方法，指定else一定要出现或者if后面加个end if

d. 人工干预

在LL(1)分析时，如果有冲突，则1个格子里可能出现2个文法，需要人工定义规则消除。

e. 最长串匹配原则

SLR(1)的分析方法，移进和规约产生冲突时，永远只保持移进，不产生规约。

6. 语法制导翻译的方法有多少种（4种，严格上来说只有3种）

- 递归子程序，添加翻译的工作（形成中间代码）
- ✨ LR的方法，规约后产生中间代码（属性文法的方法）
- LL(1)方法，推导后产生中间代码

二、正则表达式->DFA分析题

- 国手撕实验代码 (by 翁行)

1. 正则转NFA

数据结构

a. NFA图的结点

- 状态编号 `state`，默认为0
- 是否终结结点 `isEnd`，默认为 `false`
- 当前结点上的转移关系 `transfers`

```
1 struct NfaNode {
2     int state = 0;
3     bool isEnd = false;
4     map<char, vector<NfaNode*>> transfers;
5
6     NfaNode() : state(0) {}
7     NfaNode(int state) : state(state) {}
8     NfaNode(const NfaNode& node) : state(node.state), isEnd(node.isEnd), transfers(node.transfers)
9 };
```

`transfers` 指示了从某一个 `symbol`（也就是某一个字符）转移到的下一个NFA结点列表，包括 `EPSILON`，同时因为NFA允许同一字符转移到多个NFA状态，所以是列表。

b. NFA图

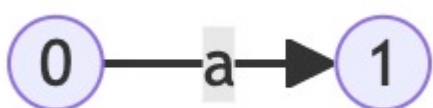
- 一个NFA子图只需保管其开始结点和终结结点，后续的联结、或、闭包等操作都是基于一个或两个NFA子图来操作的，无非是对某NFA子图的 `start` 结点或 `end` 结点进行新增 `transfers` 操作。同时，由于可能会根据NFA子图生成新的NFA子图（参照Thompson算法），所以还需要在生成新子图时更新旧子图持有的NFA结点的 `state`，也就是更新编号。

- 每一个NFA子图在创建之初的编号都默认是 {start: 0, end: 1}，随着NFA的扩张，调用 updateState 方法不断更新子图的编号。

```
1 struct NfaGraph {
2     NfaNode* start;
3     NfaNode* end;
4
5     NfaGraph(NfaNode* start = 0, NfaNode* end = 0) : start(start), end(end) {}
6     NfaGraph(const NfaGraph& graph) : start(graph.start), end(graph.end) {}
7
8     // 更新节点状态
9     void updateState(int offset) {
10         map<int, int> visited; // 已访问
11         stack<NfaNode*> prepared; // 待遍历的栈
12         vector<NfaNode*> ready; // 已遍历的结点
13         prepared.push(start); // 加入起始结点
14         while (prepared.size()) {
15             NfaNode* cur = prepared.top();
16             prepared.pop();
17             visited[cur->state] = 1; // 标记已访问
18             ready.push_back(cur);
19             for (auto& p : cur->transfers)
20                 for (NfaNode* next : p.second)
21                     if (!visited[next->state])
22                         prepared.push(next); // 将未访问过的结点加入待遍历栈
23         }
24         // 执行更新
25         for (NfaNode* node : ready)
26             node->state += offset;
27     }
28 };
```

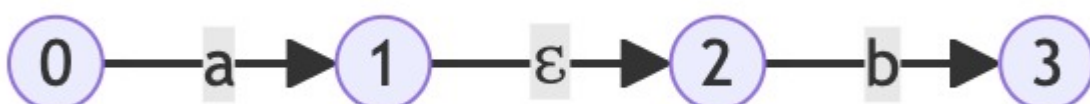
实现代码

a. 从Symbol生成NFA子图



```
1 NfaGraph fromSymbol(char symbol) { // 根据Symbol生成Nfa子图
2     NfaGraph graph(new NfaNode, new NfaNode(1));
3     graph.end->isEnd = true;
4     graph.start->transfers[symbol].push_back(graph.end);
5     return graph;
6 }
```

b. 执行连接操作



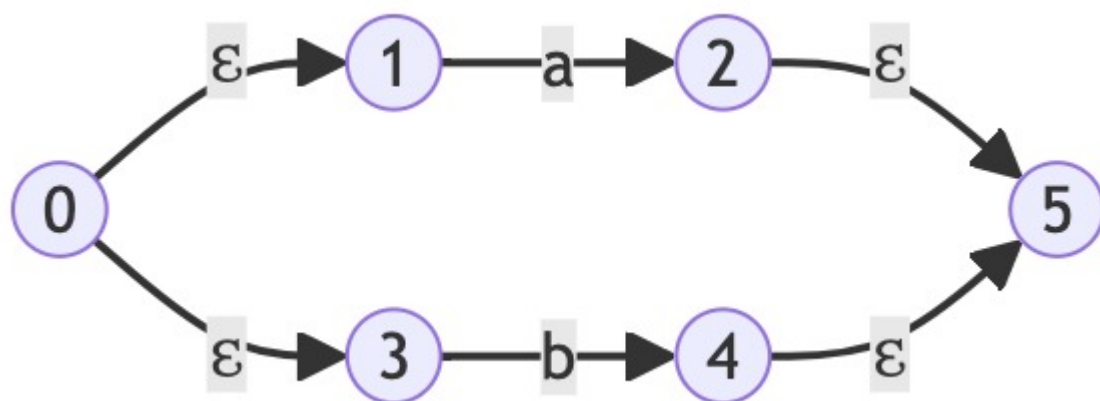
```
1 NfaGraph setConcat(NfaGraph& t1, NfaGraph& t2) { // 连接Nfa子图
```

```

2   NfaGraph graph;
3   graph.start = t1.start;
4   graph.end = t2.end;
5   t1.end->isEnd = false;
6   // 更新t2子图的结点编号
7   t2.updateState(t1.end->state + 1);
8   // 加入EPSILON转移
9   t1.end->transfers[EPSILON].push_back(t2.start);
10  return graph;
11 }

```

c. 执行或 | 操作

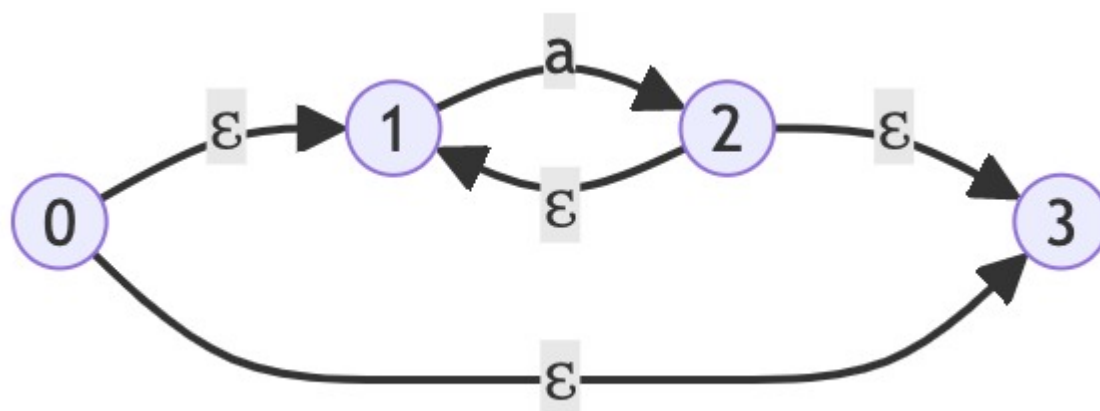


```

1  NfaGraph setUnions(NfaGraph& t1, NfaGraph& t2) { // Nfa子图 或运算
2      NfaGraph graph(new NfaNode, new NfaNode);
3      graph.end->isEnd = true;
4      // 更新子图t1、t2的结点编号
5      t1.updateState(1); // t1编号整体+1 (多了一个初始节点)
6      t2.updateState(t1.end->state + 1); // t2编号整体+t1更新后的end编号+1
7      graph.end->state = t2.end->state + 1;
8      // 取消原有的终结结点
9      t1.end->isEnd = false;
10     t2.end->isEnd = false;
11     // 加入EPSILON转移
12     graph.start->transfers[EPSILON].push_back(t1.start);
13     graph.start->transfers[EPSILON].push_back(t2.start);
14     t1.end->transfers[EPSILON].push_back(graph.end);
15     t2.end->transfers[EPSILON].push_back(graph.end);
16     return graph;
17 }

```

d. 执行闭包操作



```

1  NfaGraph setClosure(NfaGraph& target) { // Nfa子图闭包
2      NfaGraph graph(new NfaNode, new NfaNode);

```



```

3     graph.end->isEnd = true;
4     // 更新子图target的结点编号
5     target.updateState(1);
6     graph.end->state = target.end->state + 1;
7     // 取消原有终结结点
8     target.end->isEnd = false;
9     // 加入EPSILON转移
10    graph.start->transfers[EPSILON].push_back(target.start);
11    graph.start->transfers[EPSILON].push_back(graph.end);
12    target.end->transfers[EPSILON].push_back(target.start);
13    target.end->transfers[EPSILON].push_back(graph.end);
14    return graph;
15 }

```

e. 根据不同的OP，对子图执行不同的操作

- 闭包：单目运算
- 其他：双目运算

接收参数 `op` 和当前已有的子图栈 `subgraphs`，从栈里pop出相应个数的子图，对其施加相应OP的操作，再将结果push回子图栈。

```

1 void setAction(char op, stack<NfaGraph>& subgraphs) {
2     NfaGraph result; // 执行结果
3     if (op == CLOSURE) {
4         // 单目运算
5         NfaGraph& target = subgraphs.top();
6         subgraphs.pop();
7         result = setClosure(target);
8     }
9     else {
10        // 双目运算符
11        NfaGraph& t2 = subgraphs.top();
12        subgraphs.pop();
13        NfaGraph& t1 = subgraphs.top();
14        subgraphs.pop();
15        if (op == CONCAT) result = setConcat(t1, t2);
16        else result = setUnion(t1, t2);
17    }
18    // 压入结果栈
19    subgraphs.push(result);
20 }

```

f. 生成顶层NFA图

其实就是一个扫描输入字符，然后逐一新增NFA子图，根据OP对子图栈里的子图不断操作直到栈中只剩下一个NFA子图的过程。（类似于算术表达式的算法）

归纳为几点：

- i. 初始化两个栈：**操作符栈**和**子图栈**
- ii. 对输入的字符串预处理，**加入显式的连接符**。（比如 `ab` 转化为 `a.b`）
- iii. 对于预处理后的每一个字符：
 1. 左括号：压入操作符栈，扫描下一个字符；
 2. 右括号：将操作符栈的上一个左括号前的所有操作都清空，扫描下一个字符；
 3. 操作符：将操作符栈中已有的且**优先级比当前扫描到的操作符高**的操作符清空，遇到优先级比自己低的才停下，然后将自己压入操作符栈，扫描下一个字符；

4. 普通符号：根据这个Symbol生成NFA子图，压入子图栈中，同时加入Symbols的统计中（这样才能统计出所有转移字符）

iv. 扫描完预处理字符串后，清空符号栈

v. 获取到子图栈中的唯一子图，就是所生成的最终的NFA图



这里符号栈的“清空”操作意味着：从符号栈里弹出，调用 `setAction` 方法完成对NFA子图的操作。

```
1 // 生成顶层NFA图
2 void generate(string input) {
3     string prepared = ""; // 预处理后的输入字符串
4     stack<char> ops; // 操作符号栈
5     stack<NfaGraph> subgraphs; // 子图栈
6     for (int i = 0; i < input.size(); ++i) { // 预处理输入字符串（加入UNION字符）
7         char id = input[i];
8         if (skip(id)) continue; // 跳过无意义字符
9         prepared.push_back(id);
10        if (i + 1 >= input.size() ||
11            input[i + 1] == RBRACKET ||
12            input[i] == CONCAT ||
13            input[i + 1] == CONCAT ||
14            input[i] == LBRACKET ||
15            input[i + 1] == CLOSURE) continue; // 这些情况不用手动加入联结符号
16        // 人为加入表示UNION的字符
17        prepared.push_back(UNION);
18    }
19    for (int i = 0; i < prepared.size(); ++i) {
20        char id = prepared[i]; // 当前Identifier
21        if (id == LBRACKET) { // 左括号
22            ops.push(id); // 入符号栈
23            continue;
24        }
25        if (id == RBRACKET) { // 右括号
26            while (ops.size()) { // 清空和其最近匹配的左括号内的所有操作
27                char op = ops.top();
28                ops.pop();
29                if (op == LBRACKET) break; // 匹配到左括号，退出
30                setAction(op, subgraphs); // 执行操作
31            }
32            continue;
33        }
34        if (reservedSymbol(id)) { // 保留字符（运算符）
35            while (ops.size()) { // 清空符号栈里优先级比当前高的运算
36                char op = ops.top();
37                if (privilege(id) > privilege(op)) break; // 优先级没当前OP高
38                ops.pop(); // 优先级较高，出栈并执行
39                setAction(op, subgraphs); // 执行操作
40            }
41            ops.push(id); // 将当前OP压入栈
42            continue;
43        }
44        // 普通Symbol，生成子图
45        NfaGraph subgraph = fromSymbol(id);
46        subgraphs.push(subgraph);
47        symbols.insert(id); // 加入Symbol统计中
48    }
49    // 清空符号栈
50    while (ops.size()) {
51        char op = ops.top();
```

```

52     ops.pop();
53     setAction(op, subgraphs);
54 }
55 this->graph = subgraphs.top(); // 栈顶就是顶层NFA图
56 }

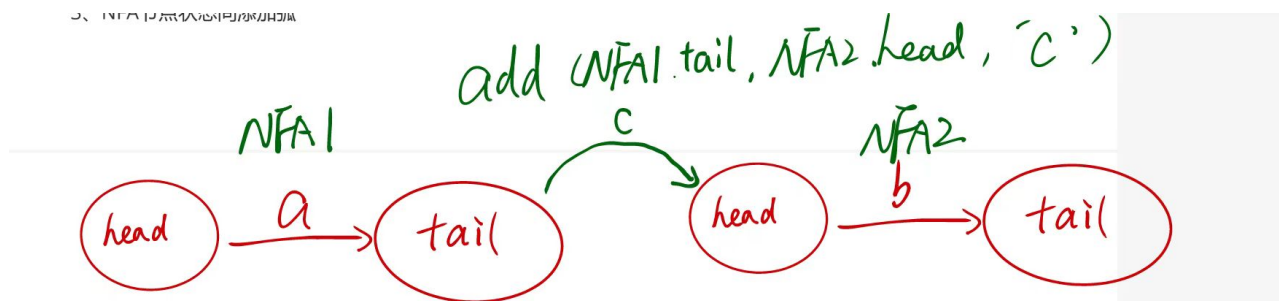
```

另一种思路，采用先中缀转后缀再处理后缀表达式（by：邹锐城）

借鉴的是往届师兄的实验二源代码，只保留了核心的

这个感觉比较容易理解？

NFA图存储结构



```

1 //NFA节点结构体
2 typedef struct NFANode
3 {
4     int id = -1; //NFA节点状态号
5     char c = '#'; //NFA节点转换弧上的值
6     int to = -1; //NFA节点转换弧转移到的状态号
7     set<int> same; //当前状态通过ε转移到的状态号集合
8 }NN;
9 //NFA结构体
10 typedef struct NFA
11 {
12     NN* h = 0; //NFA的头指针
13     NN* t = 0; //NFA的尾指针
14 }N;
15 //从状态n1到状态n2添加一条弧，弧上的值为ch
16 void add(NN* n1, NN* n2, char ch)
17 {
18     n1->c = ch;
19     n1->to = n2->id;
20 }
21
22 //从状态n1到状态n2添加一条弧，弧上的值为ε
23 void add(NN* n1, NN* n2)
24 {
25     n1->same.insert(n2->id);
26 }
27 stack<NFA> NfaStack; //NFA栈
28 int NNum; //节点数
29 void reg2Nfa(){
30     init();
31     s = infixToSuffix(s); //中缀转为后缀,s存储了正则表达式
32     while (i < length)
33     {
34         if (isalpha(s[i]) || s[i] == '+' || s[i] == '-' || s[i] == '.')
35             //如果是操作数
36         {
37             N n = creatNFA(NNum); //新建NFA
38             add(n.h, n.t, s[i]); //头到尾的s[i]转换

```

```

38         NfaStack.push(n); //将NFA加
入栈中
39     }
40     else if (s[i] == '*') //如果是闭
包运算符
41     {
42         N n = creatNFA(NNum); //新建NFA
43         N n1 = NfaStack.top(); //从栈中取出一个NFA
44         NfaStack.pop();
45         add(n.h, n.t); //n的头到n的尾的ε转换
46         add(n.h, n1.h); //n的头到n1的头的ε转
换
47         add(n1.t, n.t); //n1的尾到n的头的ε转
换
48         add(n1.t, n1.h); //n1的尾到n的头的ε转
换
49         NfaStack.push(n); //将NFA加
入栈中
50     }
51     else if (s[i] == '|') //如果是或
运算符
52     {
53         N n = creatNFA(NNum); //新建NFA
54         N n1 = NfaStack.top(); //从栈中取出两个NFA
55         NfaStack.pop();
56         N n2 = NfaStack.top();
57         NfaStack.pop();
58         add(n.h, n1.h); //n的头到n1的头的ε转
换
59         add(n.h, n2.h); //n的头到n2的头的ε转
换
60         add(n1.t, n.t); //n1的尾到n的尾的ε转
换
61         add(n2.t, n.t); //n2的尾到n的尾的ε转
换
62         NfaStack.push(n); //将NFA加
入栈中
63     }
64     else if (s[i] == '&')
65     {
66         N n;
67         N n2 = NfaStack.top(); //从栈中取出两个NFA
68         NfaStack.pop();
69         N n1 = NfaStack.top();
70         NfaStack.pop();
71         add(n1.t, n2.h); //n1的尾到n2的头的ε
转换
72         n.h = n1.h; //n的头为n1的头
73         n.t = n2.t; //n的尾为n2的尾
74         NfaStack.push(n); //将NFA加
入栈中
75     }
76     else if(s[i] == '?')
77     {
78         N n = creatNFA(NNum); //新建NFA
79         N n1 = NfaStack.top();
80         NfaStack.pop();
81         add(n.h, n1.h);
82         add(n.h, n.t);
83         add(n1.t, n.t);
84         NfaStack.push(n);
85     }

```

```
86         i++;
87     }
88     return NfaStack.top();
89 }
```

2. NFA转DFA

数据结构

a. NFA

i. NfaNode

```
1 struct NfaNode {
2     int state;
3     bool isEnd;
4     map<char, vector<NfaNode*>> transfers;
5 };
```

ii. NfaGraph

```
1 struct NfaGraph {
2     NfaNode* start;
3     NfaNode* end;
4 };
```

iii. 符号集

```
1 set<char> symbols
```

b. DFA结点

```
1 struct DfaNode {
2     int state;
3     bool isEnd = false;
4     map<char, int> transfers; // 转移
5     set<NfaNode*> nfaNodes;
6     DfaNode() : state(0) {}
7     DfaNode(int state) : state(state) {}
8     DfaNode(const DfaNode& node) : state(node.state), transfers(node.transfers),
    nfaNodes(node.nfaNodes) {}
9
10    void bindNfaNodes(set<NfaNode*> nodes) { // 将NFA集合绑定进该DFA结点, 同时更新isEnd的值
11        nfaNodes = nodes;
12        for (NfaNode* node : nodes) {
13            if (node->isEnd) {
14                isEnd = true;
15                return;
16            }
17        }
18    }
19
20    bool operator==(const DfaNode& node) { // 通过比较持有的NFA结点集合, 判断两个DFA状态是否等价
21        if (nfaNodes.size() != node.nfaNodes.size()) return false;
```

```

22     for (NfaNode* nfaNode : nfaNodes) {
23         if (!node.nfaNodes.count(nfaNode)) return false;
24     }
25     return true;
26 }
27 };

```

c. DFA图

简简单单的 `vector<DfaNode*>`，每个 `DfaNode` 的状态编号和其所在的下标是一样的。

实现代码

子集构造法，有以下几个关键点：

- 初始状态是NFA初始节点的EPSILON闭包
- 对于所有的Symbols，对已确定的DFA结点进行转移（转移到一个 `set<NfaNode*>`），判断该转移到的目标是否已经存在于已确定的DFA结点中，没有的话就新增一个DFA状态。对于该转移目标，记录进当前DFA状态的转移表中。
- 对于转移：在某**NFA结点**上，先根据某Symbol进行一次移进，同时移进结果求EPSILON闭包。

那么代码由三部分构成：EPSILON闭包 `epsilonClosure`、移进 `forward`、生成DFA `generate`

a. 求EPSILON闭包

```

1  set<NfaNode*> epsilonClosure(NfaNode* source) { // NFA节点的EPSILON闭包
2      set<NfaNode*> closure;
3      stack<NfaNode*> prepared; // DFS栈
4      map<int, int> visited;
5      prepared.push(source);
6      while (prepared.size()) {
7          NfaNode* cur = prepared.top();
8          prepared.pop();
9          if (visited[cur->state]) continue;
10         closure.insert(cur);
11         visited[cur->state] = 1;
12         for (NfaNode* next : cur->transfers[EPSILON])
13             prepared.push(next);
14     }
15     return closure;
16 }

```

b. 根据某Symbol进行一次移进，并对结果求EPSILON闭包

```

1  set<NfaNode*> forward(NfaNode* source, char symbol) { // 以symbol步进
2      set<NfaNode*> result;
3      for (NfaNode* next : source->transfers[symbol]) {
4          result.insert(next);
5          // 同时要加入其EPSILON闭包
6          set<NfaNode*> closureOfNext = epsilonClosure(next);
7          result.insert(closureOfNext.begin(), closureOfNext.end());
8      }
9      return result;
10 }

```

c. 生成DFA

```

1 // NfaGraph nfaGraph

```

```

2 // set<char> symbols
3 void generate() { // 生成DFA图
4     DfaNode* start = new DfaNode();
5     start->bindNfaNodes(epsilonClosure(nfaGraph.start)); // 将NFA节点列表绑定进DFA状态中
6     nodes.push_back(start); // 加入初始节点
7     for (int i = 0; i < nodes.size(); ++i) {
8         DfaNode* cur = nodes[i];
9         for (char symbol : symbols) {
10             set<NfaNode*> nfaNodesOfSymbol;
11             for (NfaNode* next : cur->nfaNodes) {
12                 set<NfaNode*> nfaNodesOfNext = forward(next, symbol);
13                 nfaNodesOfSymbol.insert(nfaNodesOfNext.begin(), nfaNodesOfNext.end());
14             }
15             if (nfaNodesOfSymbol.empty()) continue;
16             // 判断是否新增
17             DfaNode* instance = new DfaNode(nodes.size());
18             instance->bindNfaNodes(nfaNodesOfSymbol);
19             // 子集构造法
20             for (DfaNode* exist : nodes) {
21                 if (*exist == *instance) {
22                     instance = exist;
23                     break;
24                 }
25             }
26             if (instance->state == nodes.size()) { // 需要新增节点
27                 nodes.push_back(instance);
28             }
29             // 加入转移关系
30             cur->transfers[symbol] = instance->state;
31         }
32     }
33 }

```

3. DFA最小化（分拆方式）

数据结构

a. MDFA结点

- 状态编号 `state`
- 是否终结结点 `isEnd`，由所持有的DFA结点决定
- 所持有的DFA结点集合 `dfaNodes`
- 当前状态下的转移关系 `transfers`

```

1 struct MDfaNode {
2     int state;
3     bool isEnd;
4     set<DfaNode*> dfaNodes; // 持有的DFA结点
5     map<char, int> transfer;
6
7     void bindDfaNodes(set<DfaNode*> dfaNodes) {
8         this->dfaNodes = dfaNodes;
9         for (DfaNode* node : dfaNodes) {
10             if (node->isEnd) {
11                 isEnd = true;
12                 break;
13             }
14         }
15     }
16 }

```

```

16
17     MDfaNode(int state) : state(state), isEnd(false) {}
18
19     bool operator==(const MDfaNode& node) { // 根据所持有的DFA结点，比较二者是否等价
20         if (node.dfaNodes.size() != dfaNodes.size()) return false;
21         for (DfaNode* dfaNode : node.dfaNodes)
22             if (!dfaNodes.count(dfaNode)) return false;
23         return true;
24     }
25 };

```

b. MDFA图

简简单单的 `vector<MDdfaNode*>` nodes

实现代码

a. 从一个 `set<DfaNode*>`，根据某Symbol移进的状态集合，用于根据MDFA生成转移关系

```

1  set<int> forward(set<DfaNode*> source, char symbol) {
2      set<int> result;
3      for (DfaNode* node : source) {
4          if (!node->transfers.count(symbol)) continue; // 当前DFA结点上不存在该转移关系
5          result.insert(node->transfers[symbol]);
6      }
7      return result;
8  }

```

b. DFA最小化

- 已拆分列表
- 待拆分列表
- 暂存某状态集合中每个状态转移目的地的map（用于拆分）
- 顶层循环是**Symbol**，每一次进入循环，都以上一次的已拆分结果作为本次的待拆分目标。

```

1  // set<char> symbols
2  void minimize() { // 最小化
3      set<DfaNode*> left, right; // 两个拆分
4      vector<set<DfaNode*>> completed; // 已完成拆分
5      vector<set<DfaNode*>> prepared; // 待拆分
6      map<int, set<DfaNode*>> destination; // 转移目的地，做暂存使用
7      // 根据是否终结节点初始化
8      for (DfaNode* node : dfa.getNodes()) {
9          if (node->isEnd) right.insert(node);
10         else left.insert(node);
11     }
12     completed.push_back(left);
13     completed.push_back(right);
14     for (const char symbol : symbols) {
15         prepared = completed; // 将上一次的拆分结果作为这一次的待拆分队列
16         completed.clear(); // 清空拆分结果
17         while (prepared.size()) {
18             if (prepared.empty()) break;
19             destination.clear();
20             set<DfaNode*> cur = prepared.front();
21             if (cur.empty()) {

```

```

22         prepared.erase(prepared.begin(), prepared.begin() + 1); // 把cur出队
23         continue; // 空集合
24     }
25     if (cur.size() == 1) {
26         completed.push_back(cur); // 长度为1, 无需拆分
27         prepared.erase(prepared.begin(), prepared.begin() + 1);
28         continue;
29     }
30     for (DfaNode* node : cur) {
31         if (!node->transfers.count(symbol)) { // 不存在该转移
32             destination[-1].insert(node);
33             continue;
34         }
35         int target = node->transfers[symbol]; // 下一个DFA状态
36         for (int i = 0; i < prepared.size(); ++i) {
37             bool matched = false;
38             for (DfaNode* state : prepared[i]) {
39                 if (state->state == target) { // 移进的目标是这个集合
40                     matched = true;
41                     destination[i].insert(node);
42                     break;
43                 }
44             }
45             if (matched) break;
46         }
47         for (int i = 0; i < completed.size(); ++i) {
48             bool matched = false;
49             for (DfaNode* state : completed[i]) {
50                 if (state->state == target) { // 移进的目标是这个集合
51                     matched = true;
52                     destination[prepared.size() + i].insert(node);
53                     break;
54                 }
55             }
56             if (matched) break;
57         }
58     }
59     prepared.erase(prepared.begin(), prepared.begin() + 1); // 把cur出队
60     // 找内奸
61     if (destination.size() > 1) { // 有内奸
62         for (auto& p : destination)
63             prepared.push_back(p.second);
64         continue;
65     }
66     // 没有内奸
67     completed.push_back(cur);
68 }
69 }
70 // 根据划分结果生成MDFA结点
71 MDfaNode* startNode = new MDfaNode(0);
72 set<DfaNode*> divideOfStartNode;
73 for (set<DfaNode*> divide : completed) {
74     for (DfaNode* node : divide) {
75         if (node->state == 0) {
76             divideOfStartNode = divide;
77             break;
78         }
79     }
80     if (divideOfStartNode.size()) break; // 已经找到起始划分
81 }
82 startNode->bindDfaNodes(divideOfStartNode); // 绑定划分
83 nodes.push_back(startNode);

```



```

84     for (int i = 0; i < nodes.size(); ++i) { // 生成MDFA结点
85         MDfaNode* cur = nodes[i];
86         for (const char symbol : symbols) {
87             set<int> next = forward(cur->dfaNodes, symbol); // 获取转移目标
88             if (next.empty()) continue;
89             for (set<DfaNode*> divide : completed) { // 寻找和目标等价的划分
90                 bool matched = false;
91                 for (DfaNode* node : divide) {
92                     if (next.count(node->state)) {
93                         matched = true;
94                         break;
95                     }
96                 }
97                 if (!matched) continue; // 不匹配
98                 MDfaNode* instance = new MDfaNode(nodes.size());
99                 instance->bindDfaNodes(divide);
100                for (MDfaNode* exist : nodes) { // 寻找是否已经存在该结点
101                    if (*exist == *instance) {
102                        instance = exist;
103                        break;
104                    }
105                }
106                if (instance->state == nodes.size()) // 不存在的结点
107                    nodes.push_back(instance);
108                cur->transfer[symbol] = instance->state;
109                break;
110            }
111        }
112    }
113 }

```

另一种思路，采用hopcroft算法（by：邹锐城）

hopcroft算法是浴帘上课的手工算法名字，因为采用了状态压缩方法，所以代码量会少很多（不太好理解，但胜在代码量少，不懂的抄两遍记住就好），存储结构核心代码量大概只有73行

参考博客：

[【编译原理实验】Lab2\(二\)DFA最小化_dfa最小化代码实现-CSDN博客](#)

[DFA 的最小化——详解 Hopcroft 算法](#)

核心代码：

```

1  struct Node{//DFA节点
2      int id;
3      bool flag;//标记终态
4  };
5  struct Edge{//DFA边
6      int to;
7      char c;
8  };
9  char z fj[101];//字符集合
10 int z fnum;
11 vector<Node> V;//DFA_Vertex集
12 vector<Edge> edge[101];//DFA边集
13 int v,e;//顶点数和边数
14 int stateNum = 2;//默认分为终态和非终态
15 vector<int> eq;//等价状态集合
16 int p[101]; //记录状态所属集合
17 void split(int x,char c)
18 {

```

```

19     int s = eq[x];
20     int a[101]={0}; //吃字符达到的状态
21     int b[101]={0}; //吃字符前的状态
22     int i,j,k;
23     for(i = 0;i < v;i++){
24         if((s>>i)&1){ //遍历集合中的状态
25             for(j = 0;j < edge[i].size();j++){
26                 if(edge[i][j].c == c){
27                     int v = edge[i][j].to;
28                     a[p[v]] |= (1<<v);
29                     b[p[v]] |= (1<<i);
30                 }
31             }
32         }
33     }
34     int ll = stateNum;
35     for(i = 0;i < ll;i++){
36         if(b[i]==eq[x]) break;
37         if(a[i] && i!=x){
38             eq.push_back(b[i]);
39             for(k = 0;k < v;k++){
40                 if((b[i]>>k)&1)
41                     p[k] = stateNum;
42             eq[x]&=(~b[i]);
43             stateNum++;
44         }
45     }
46 }
47 void Hopcroft()
48 {
49     int N = 0,T = 0;
50     for(int i = 0;i < v;i++){ //分成N和T两个集合
51         if(V[i].flag){
52             T |= (1<<V[i].s);
53             p[V[i].s] = 1;
54         }
55         else{
56             N |= (1 << V[i].s);
57             p[V[i].s] = 0;
58         }
59     }
60     if(N == 0) return ;
61     eq.push_back(N);
62     eq.push_back(T);
63     int i,j;
64     int l;
65     while(1){
66         for(i = 0;i < zfnun;i++){
67             l = stateNum;
68             for(j = 0;j < l;j++){
69                 split(j,zfj[i]);
70             }
71             if(l==t) break; //集合大小不变算法结束
72         }
73     }
74     for(int i = 0;i < v;i++) //合并边集
75         for(int j = 0;j < edge[i].size();j++)
76             Ed.insert(Edge_dfa(p[i],p[edge[i][j].to],edge[i][j].c));

```

4. DFA最小化（合并方式）

hyl视频说效率不高，没讲具体算法，如果有代码欢迎贡献

5. 词法生成器生成代码

存储结构：

```
1 struct dfaMinNode
2 {
3     string flag; // 是否包含终态 (+) 或初态 (-)
4     int id;
5     map<char, int> transitions; // 字符到下一状态的映射
6     dfaMinNode() {
7         flag = "";
8     }
9 };
10 vector<dfaMinNode> dfaMinTable;
```

考场上，下面codeStream换成cout就好了。。。就不用 resultCode 这个变量了

```
1 string resultCode;
2 // 生成词法分析器代码并返回为字符串
3 void generateLexerCode() {
4     ostringstream codeStream;
5
6     codeStream << "#include <iostream>" << endl;
7     codeStream << "#include <string>" << endl;
8     codeStream << endl;
9     codeStream << "using namespace std;" << endl;
10    codeStream << endl;
11    codeStream << "int main() {" << endl;
12    codeStream << "    string input;" << endl;
13    codeStream << "    cin >> input;" << endl;
14    codeStream << "    int currentState = 0;" << endl;
15    codeStream << "    int length = input.length();" << endl;
16    codeStream << "    for (int i = 0; i < length; i++) {" << endl;
17    codeStream << "        char c = input[i];" << endl;
18    codeStream << "        switch (currentState) {" << endl;
19
20        for (const dfaMinNode& node : dfaMinTable) {
21            codeStream << "            case " << node.id << ":" << endl;
22            codeStream << "                switch (c) {" << endl;
23            for (const auto& transition : node.transitions) {
24                if (transition.second != -1) {
25                    codeStream << "                    case '" << transition.first << ":" << endl;
26                    codeStream << "                        currentState = " << transition.second << ";" <<
endl;
27                }
28                codeStream << "                    break;" << endl;
29            }
30            codeStream << "                default:" << endl;
31            codeStream << "                    cout << \"Error: Invalid input character '\" << c <<
\"\"\" << endl;" << endl;
32            codeStream << "                    return 1;" << endl;
33            codeStream << "                }" << endl;
34            codeStream << "            }" << endl;
35        }
36
37        codeStream << "    }" << endl;
38    codeStream << "}" << endl;
```

```

39     codeStream << "        switch (currentState) {" << endl;
40
41     for (const dfaMinNode& node : dfaMinTable) {
42         if (node.flag.find("+") != string::npos) {
43             codeStream << "            case " << node.id << ":" << endl;
44             codeStream << "                cout << \"Accepted\" << endl;" << endl;
45             codeStream << "                break;" << endl;
46         }
47     }
48
49     codeStream << "        default:" << endl;
50     codeStream << "            cout << \"Not Accepted\" << endl;" << endl;
51     codeStream << "    }" << endl;
52     codeStream << "    return 0;" << endl;
53     codeStream << "}" << endl;
54
55     resultCode = codeStream.str();
56 }

```

三、自顶向下分析题

本年度实验没做消除左递归和左公因子

猜测题型如下：（也可能是多个题型结合在一起）

1. 题型一

三、自顶向下分析设计题（1题，共17分）

1. 如果要采用递归下降分析方法（或称递归子程序分析法）生成逻辑表达式对应的语法树，那我们需要解决的问题有：

- （1）定义逻辑表达式的文法规则；
- （2）逻辑表达式语法树的存储结构；
- （3）写出文法规则对应的语法树生成算法。

现请你解决以上的三个问题。

TINY实验三完整规则：

重点记一下for语句（黄色）、位运算（红色）、正则部分（已被AI考了）的语法规则

```

1  program->stmt-sequence
2  stmt-sequence->stmt-sequence;statement | statement
3  statement->if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt | plusassign-stmt | for-to-stmt |
   for-downto-stmt | regex-stmt
4  if-stmt-->if(exp) [ stmt-sequence [[ else stmt-sequence]] ] （里面的[]代表不是EBNF语法的[]）
5  repeat-stmt->repeat stmt-sequence until exp
6  assign-stmt->identifier := exp
7  plusassign-stmt->identifier += exp
8  read-stmt->read identifier
9  write-stmt->write exp
10 for-to-stmt-->for identifier:=simple-exp to simple-exp do stmt-sequence enddo
11 for-downto-stmt--> for identifier:=simple-exp downto simple-exp do stmt-sequence enddo
12 exp -> exp orop oexp | oexp
13 orop -> or
14 oexp -> oexp andop andexp | andexp
15 andop -> and
16 andexp -> simple-exp comparison-op simple-exp | simple-exp
17 comparison-op -> < | > | <= | >= | <>
18 simple-exp -> simple-exp addop term | term
19 addop -> + | -
20 term -> term mulop notexp | notexp

```

```

21 mulop -> * | / | %
22 notexp -> notop notexp | power
23 notop -> not
24 power -> power powop factor | factor
25 powop -> ^
26 factor -> (exp) | number | identifier
27 ----- (正则部分) -----
28 regex-stmt->identifier ::= regex_exp
29 regex_exp-> regex_exp rorop andreg | andreg
30 rorop -> |
31 andreg -> andreg randop topreg | topreg
32 randop -> &
33 topreg -> topreg topop | reg_factor
34 topop -> # | ?
35 reg_factor -> (regex_exp) | ideifier | number

```

语法树存储结构（通用）：

标黄为自行新增的

```

1 typedef enum { StmtK, ExpK } NodeKind;
2 typedef enum { IfK, RepeatK, AssignK, ReadK, WriteK, ForToK, ForDowntoK, RegexK } StmtKind;
3 typedef enum { OpK, ConstK, IdK } ExpKind;
4
5 /* ExpType is used for type checking */
6 typedef enum { Void, Integer, Boolean } ExpType;
7
8 #define MAXCHILDREN 3
9
10 typedef struct treeNode
11 {
12     struct treeNode* child[MAXCHILDREN];
13     struct treeNode* sibling;
14     NodeKind nodekind;
15     union { StmtKind stmt; ExpKind exp; } kind;
16     union {
17         TokenType op;
18         int val;
19         char* name;
20     } attr;
21     ExpType type; /* for type checking of exps */
22 } TreeNode;

```

1. ForToK, ForDowntoK为例子，生成算法如下：

```

1 TreeNode* for_stmt(void)
2 {
3     // 赋值节点
4     TreeNode* p = newStmtNode(AssignK);
5     match(FOR);
6     if ((p != NULL) && (token == ID))
7     {
8         p->attr.name = copyString(tokenString);
9     }
10    match(ID);
11    match(ASSIGN);
12    if (p != NULL)
13    {

```

```

14     p->child[0] = simple_exp();
15 }
16 // FOR节点
17 TreeNode* t = NULL;
18 if (token == T0) { // 步长+1
19     t = newStmtNode(ForToK);
20     t->child[0] = p;
21     match(T0);
22 }
23 else if (token == DOWNT0) { // 步长-1
24     t = newStmtNode(ForDowntoK);
25     t->child[0] = p;
26     match(DOWNT0);
27 }
28 else // 出错提示
29 {
30     syntaxError("Expecting 'to' or 'downto' after assignment in for statement");
31 }
32 t->child[1] = simple_exp();
33 match(D0);
34 t->child[2] = stmt_sequence();
35 match(ENDDO);
36 return t;
37 }

```

2. 正则表达式

文法如下（EBNF）：

```

1 regex_stmt -> andreg {orop andreg }
2 orop -> |
3 andreg -> topreg {andop topreg}
4 andop -> &
5 topreg -> reg_factor {topop}
6 topop -> # | ?
7 reg_factor -> (regex_stmt ) | ideifier | number

```

对于花括号来说，我们使用while来解析，对于[]就是用if语句

```

1 TreeNode* regex_stmt(void)
2 {
3     TreeNode* t = andreg();
4     while ((token == RGOR))
5     {
6         TreeNode* p = newExpNode(OpK);
7         if (p != NULL) {
8             p->child[0] = t;
9             p->attr.op = token;
10            t = p;
11            match(token);
12            t->child[1] = andreg();
13        }
14    }
15    return t;
16 }
17
18 TreeNode* andreg(void)
19 {

```

```

20     TreeNode* t = topreg();
21     while ((token == RGAND))
22     {
23         TreeNode* p = newExpNode(OpK);
24         if (p != NULL) {
25             p->child[0] = t;
26             p->attr.op = token;
27             t = p;
28             match(token);
29             t->child[1] = topreg();
30         }
31     }
32     return t;
33 }
34
35 TreeNode* topreg(void)
36 {
37     TreeNode* t = reg_factor();
38     while ((token == RGCLOSE) || (token == RGCHOOSE))
39     {
40         TreeNode* p = newExpNode(OpK);
41         if (p != NULL) {
42             p->child[0] = t;
43             p->attr.op = token;
44             t = p;
45             match(token);
46         }
47     }
48     return t;
49 }
50
51 TreeNode* reg_factor(void)
52 {
53     TreeNode* t = NULL;
54     switch (token) {
55     case NUM:
56         t = newExpNode(ConstK);
57         if ((t != NULL) && (token == NUM))
58             t->attr.val = atoi(tokenString);
59         match(NUM);
60         break;
61     case ID:
62         t = newExpNode(IdK);
63         if ((t != NULL) && (token == ID))
64             t->attr.name = copyString(tokenString);
65         match(ID);
66         break;
67     case LPAREN: // 左括号
68         match(LPAREN);
69         t = regex_stmt();
70         match(RPAREN);
71         break;
72     default:
73         syntaxError("unexpected token -> ");
74         printToken(token, tokenString);
75         token = getToken();
76         break;
77     }
78     return t;
79 }

```

2. 题型二（重点）

求follow集合和求first集合（其中一种版本代码，另外还能参考的代码请见：[国编译原理复习提纲（浴帘🚿）](#)）

a. first集合：（非递归）

```
First(x)={ };
K=1;
While (k<=n)
{ if (xk 为终结符号或  $\epsilon$ ) first(xk)=xk;
  first(x)=first(x)  $\cup$  first(xk) - { $\epsilon$ }
  If (  $\epsilon \notin$  first(xk) ) break;
  k++;
}
If (k==n+1) first(x)=first(x)  $\cup$   $\epsilon$ 
```

```
1 // 结构化后的文法map
2 unordered_map<char, set<string>> grammarMap;
3
4 // First集合单元
5 struct firstUnit
6 {
7     set<char> s;
8     bool isEpsilon = false;
9 };
10
11 // 非终结符的First集合
12 map<char, firstUnit> firstSets;
13
14 // 计算First集合
15 bool calculateFirstSets()
16 {
17     bool flag = false;
18     for (auto& grammar : grammarMap)
19     {
20         char nonTerminal = grammar.first;
21         // 保存当前First集合的大小，用于检查是否有变化
22         size_t originalSize = firstSets[nonTerminal].s.size();
23         bool originalE = firstSets[nonTerminal].isEpsilon;
24         for (auto& g : grammar.second)
25         {
26             int k = 0;
27             while (k <= g.size() - 1)
28             {
29                 set<char> first_k;
30                 if (g[k] == '@') // @是空串（实验4要求），可改为 $\epsilon$ 
```



```

31         {
32             k++;
33             continue;
34         }
35         else if (isSmallAlpha(g[k]))
36         {
37             first_k.insert(g[k]);
38         }
39         else
40         {
41             first_k = firstSets[g[k]].s;
42         }
43         firstSets[nonTerminal].s.insert(first_k.begin(), first_k.end());
44         // 如果是终结符或者没有空串在非终结符中, 直接跳出
45         if (isSmallAlpha(g[k]) || !firstSets[g[k]].isEpsilon)
46         {
47             break;
48         }
49         k++;
50     }
51     if (k == g.size())
52     {
53         firstSets[nonTerminal].isEpsilon = true;
54     }
55 }
56 // 看原始大小和是否变化epsilon, 如果变化说明还得重新再来一次
57 if (originalSize != firstSets[nonTerminal].s.size() || originalE !=
firstSets[nonTerminal].isEpsilon)
58 {
59     flag = true;
60 }
61 }
62 return flag;
63 }
64
65 void getFirstSets()
66 {
67     // 不停迭代, 直到First集合不再变化
68     bool flag = false;
69     do
70     {
71         flag = calculateFirstSets(); // 改成递归就是把他放进子函数里面
72     } while (flag);
73 }

```

b. follow集合

1.初始化:

1.1 Follow(开始符号)={ \$ }

1.2 其他任何一个非终结符号A, 则执行 Follow(A)={ }

2.循环: 反复执行

2.1 循环: 对于文法中的每条规则 $A \rightarrow X_1 X_2 \dots X_n$ 都执行

2.1.1 对于该规则中的每个属于非终结符号的 X_i , 都执行

2.1.1.1 把 $\text{First}(X_{i+1} X_{i+2} \dots X_n) - \{\epsilon\}$ 添加到 $\text{Follow}(X_i)$

2.1.1.2 if ϵ in $\text{First}(X_{i+1} X_{i+2} \dots X_n)$, 则把Follow(A)添加到 Follow(X_i)

直到任何一个Follow集合的值都没有发生变化为止。

$$A \rightarrow X_1 X_2 \dots X_i X_{i+1} \dots X_n$$

```
1 // 结构化后的文法map
2 unordered_map<char, set<string>> grammarMap;
3 // Follow集合单元
4 struct followUnit
5 {
6     set<char> s;
7 };
8 // 非终结符的Follow集合
9 map<char, followUnit> followSets;
10 // 添加Follow集合
11 void addToFollow(char nonTerminal, const set<char>& elements)
12 {
13     followSets[nonTerminal].s.insert(elements.begin(), elements.end());
14 }
15 // 计算Follow集合
16 bool calculateFollowSets()
17 {
18     bool flag = false;
19     for (auto& grammar : grammarMap)
20     {
21         char nonTerminal = grammar.first;
22         for (auto& g : grammar.second)
23         {
24             for (int i = 0; i < g.size(); ++i)
25             {
26                 if (isSmallAlpha(g[i]) || g[i] == '@') //同理, 可以是ε
27                 {
28                     continue; // 跳过终结符
29                 }
30                 set<char> follow_k;
31                 size_t originalSize = followSets[g[i]].s.size();
32                 if (i == g.size() - 1)
33                 {
34                     // Case A: A -> αB, add Follow(A) to Follow(B)
```

```

35         follow_k.insert(followSets[nonTerminal].s.begin(),
followSets[nonTerminal].s.end());
36     }
37     else
38     {
39         // Case B:  $A \rightarrow \alpha B \beta$ 
40         int j = i + 1;
41         while (j < g.size())
42         {
43             if (isSmallAlpha(g[j]))
44             { // 终结符直接加入并跳出
45                 follow_k.insert(g[j]);
46                 break;
47             }
48             else
49             { // 非终结符加入first集合
50                 set<char> first_beta = firstSets[g[j]].s;
51                 follow_k.insert(first_beta.begin(), first_beta.end());
52
53                 // 如果没有空串在first集合中, 停止。
54                 if (!firstSets[g[j]].isEpsilon)
55                 {
56                     break;
57                 }
58                 ++j;
59             }
60         }
61         // If  $\beta$  is  $\epsilon$  or  $\beta$  is all nullable, add Follow(A) to Follow(B)
62         if (j == g.size())
63         {
64             follow_k.insert(followSets[nonTerminal].s.begin(),
followSets[nonTerminal].s.end());
65         }
66     }
67     addToFollow(g[i], follow_k);
68     // 检查是否变化
69     if (originalSize != followSets[g[i]].s.size())
70     {
71         flag = true;
72     }
73 }
74 }
75 }
76 return flag;
77 }
78 void getFollowSets()
79 {
80     // 开始符号加入$
81     addToFollow(startSymbol, { '$' });
82     // 不停迭代, 直到Follow集合不再变化
83     bool flag = false;
84     do
85     {
86         flag = calculateFollowSets();
87     } while (flag);
88 }
89

```

3. 题型三

LL(1)分析

a. 如何构建LL(1)分析表？

LL(1)分析表的构造步骤

为每个非终结符 A 和产生式 $A \rightarrow \alpha$ 重复以下两个步骤：

1) 对于 $\text{First}(\alpha)$ 中的每个记号 a ，都将 $A \rightarrow \alpha$ 添加到项目 $M[A, a]$ 中。

• 2) 若 ϵ 在 $\text{First}(\alpha)$ 中，则对于 $\text{Follow}(A)$ 的每个元素 a (记号或是 $\$$)，都将 $A \rightarrow \alpha$ 添加到 $M[A, a]$ 中。

例题：

$G[S]=\{$
 $S \rightarrow AbB \mid Bc$
 $A \rightarrow aA \mid \epsilon$
 $B \rightarrow d \mid e$
 $\}$

先求first集合：

- 1

$S \rightarrow AbB$ {a,b}
- 2

$S \rightarrow Bc$ {d,e}
- 3

$A \rightarrow aA$ {a}
- 4

$A \rightarrow \epsilon$ 此时应该求Follow集合 {b}
- 5

$B \rightarrow d$ {d}
- 6

$B \rightarrow e$ {e}

然后，集合里面有啥，就填到对应的格子里

	a	b	d	e
S	AbB	AbB	Bc	Bc

A	aA	ϵ		
B			d	e

b. 判断是否符合LL(1)文法 (本年度复习提纲新增)

- 1. 在每个产生式 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ 中, 对于所有的 i 和 $j: 1 \leq i, j \leq n, i \neq j, \text{First}(\alpha_i) \cap \text{First}(\alpha_j)$ 为空。
- 2. 若对于每个非终结符 A 都有 $\text{First}(A)$ 包含了 ϵ , 那么 $\text{First}(A) \cap \text{Follow}(A)$ 为空。

例题1:

$G[S] = \{$
 $S \rightarrow bB \mid Cc$
 $A \rightarrow aAB \mid \epsilon$
 $B \rightarrow a \mid d$
 $C \rightarrow e \mid \epsilon$
 $\}$

该例题满足第一个条件（没有左公因子），不满足第二个条件（含有 ϵ 的A的first集合{a}，follow集合为{a,d}，交集不为空），因此不符合LL(1)文法。

例题2:

$G[S] = \{$
 $S \rightarrow bB \mid ACc$
 $A \rightarrow aA \mid bB \mid \epsilon$
 $B \rightarrow e \mid d$
 $C \rightarrow f \mid \epsilon$
 $\}$

该例题不满足第一个条件（S有左公因子b），满足第二个条件，因此不符合LL(1)文法。

四、LR分析题

1. 题型一

手撕判断SLR(1)程序

存储结构：

```
1 // DFA表状态
2 struct dfaState
3 {
4     int sid; // 状态id
5     vector<int> cellV; // 存储这个状态的cellid
6     bool isEnd = false; // 是否为规约状态
7 };
8
9 // 用于通过编号快速找到对应结构
10 vector<dfaState> dfaStateVector;
11
12 // DFA表每一项项目的结构
13 struct dfaCell
14 {
15     int cellid; // 这一项的编号，便于后续判断状态相同
16     int gid; // 文法编号
17     int index = 0; // .在第几位，如i=3, xxx.x, i=0, .xxxx, i=4, xxxx
18 };
19
20 // 用于通过编号快速找到对应结构
21 vector<dfaCell> dfaCellVector;
22
23 // 文法unit (用于LR0)
24 struct grammarUnit
25 {
26     int gid;
27     char left;
28     string right;
29 };
30
31 // 文法数组 (用于LR0)
32 deque<grammarUnit> grammarDeque;
33
34 struct followUnit
35 {
36     set<char> s;
37 };
38 // 非终结符的Follow集合
39 map<char, followUnit> followSets;
```

程序：

```
1 // 检查移进-规约冲突
2 bool SLR1Fun1()
3 {
4     for (const dfaState& state : dfaStateVector)
5     {
```

```

6      // 规约项目的左边集合
7      set<char> a;
8      // 终结符
9      set<char> rVT;
10     // 不是规约状态不考虑
11     if (!state.isEnd) continue;
12     // 规约状态
13     for (int cellid : state.cellV)
14     {
15         // 拿到这个cell
16         const dfaCell& cell = dfaCellVector[cellid];
17         // 获取文法
18         const grammarUnit gm = grammarDeque[cell.gid];
19         // 判断是不是规约项目
20         if (cell.index == gm.right.length() || gm.right == "@")
21         {
22             a.insert(gm.left);
23         }
24         // 判断是不是终结符
25         else
26         {
27             if (isSmallAlpha(gm.right[cell.index]))
28             {
29                 rVT.insert(gm.right[cell.index]);
30             }
31         }
32     }
33     for (char c : a)
34     {
35         for (char v : rVT)
36         {
37             if (followSets[c].s.find(v) != followSets[c].s.end())
38             {
39                 return true;
40             }
41         }
42     }
43 }
44 }
45 return false;
46 }
47
48 bool SLR1Fun2()
49 {
50     // 检查规约-规约冲突
51     for (const auto& state : dfaStateVector)
52     {
53         // 规约项目的左边集合
54         set<char> a;
55         // 不是规约状态不考虑
56         if (!state.isEnd) continue;
57         // 规约状态
58         for (int cellid : state.cellV)
59         {
60             // 拿到这个cell
61             const dfaCell& cell = dfaCellVector[cellid];
62             // 获取文法
63             const grammarUnit gm = grammarDeque[cell.gid];
64             // 判断是不是规约项目
65             if (cell.index == gm.right.length() || gm.right == "@")
66             {
67                 a.insert(gm.left);

```

```

68     }
69 }
70 for (char c1 : a)
71 {
72     for (char c2 : a)
73     {
74         if (c1 != c2)
75         {
76             // 判断followSets[c1]和followSets[c2]是否有交集
77             set<char> followSetC1 = followSets[c1].s;
78             set<char> followSetC2 = followSets[c2].s;
79             set<char> intersection;
80             // 利用STL算法求交集
81             set_intersection(
82                 followSetC1.begin(), followSetC1.end(),
83                 followSetC2.begin(), followSetC2.end(),
84                 inserter(intersection, intersection.begin())
85             );
86             // 如果交集非空，说明存在规约-规约冲突
87             if (!intersection.empty())
88             {
89                 return true;
90             }
91         }
92     }
93 }
94 }
95 return false;
96 }
97
98 // SLR1分析
99 int SLR1Analyse()
100 {
101     bool flag1 = SLR1Fun1();
102     bool flag2 = SLR1Fun2();
103     if (flag1 && flag2)
104     {
105         return 3; // 两个冲突都有
106     }
107     else if (flag1)
108     {
109         return 1; // 移进规约冲突
110     }
111     else if (flag2)
112     {
113         return 2; // 规约规约冲突
114     }
115     // 没有冲突，是SLR(1)文法
116     return 0;
117 }

```

师兄解法：核心都有，背不下上面的可以背这个

 [2021年1月编译原理第4题.pdf](#)

对师兄解法的详细注释

```

1 struct LL { // LR(0)DFA图中“状态”和“结点”是一个意思
2     string left[N], right[N]; // right[i]、left[i]表示第i个产生式左右两部分
3     int index[N]; // index[i]表示在第i个项目中"点的位置"

```



```

4     int len; // 表示在这个L[i]状态中项目的数量，每个状态会包含这个状态下所有的项目
5 } L[M]; // 整个LR(0)DFA图中若有M个结点则有 M 个状态
6
7 bool judge() { // 判断
8     vector<int> v; // 用来存储规约项的编号
9     for (int k = 0; k < M; k++) { // 遍历当前DFA图中的所有结点
10         v.clear(); // 每开始一个结点的分析，都要把上一个结点存储下来的规约项编号清除
11         for (int i = 0; i < L[k].len; i++) {
12             if (index[i] == L[k].right[i].size()) { // 说明是规约项
13                 v.push_back(i); // 当点的位置和文法右部分的长度相同，说明点的位置到文法的最右边了 A->ab.
14                 index[i]==right[i]==2
15             }
16         }
17         // 判断:规约-规约冲突
18         for (int i = 0; i < v.size(); i++) { // 外层循环遍历当前结点的所有规约项
19             for (int j = i + 1; j < v.size(); j++) { // 内层循环的规约项与外层循环相比较
20                 // judge2是判断两个集合是否有交集 v[]返回规约项的编号，交给left[]返回文法的左部
21                 if (judge2(Follow(L[k].left[v[i]]), Follow(L[k].left[v[j]]))) {
22                     // 若两个规约项的Follow集合有交集，则不是SLR(1)文法
23                     return false;
24                 }
25             }
26         }
27         // 判断:移进-规约冲突
28         for (int i = 0; i < v.size(); i++) { // 用 v 作为索引遍历当前状态所有规约项
29             for (int j = 0; j < L[k].len; j++) { // 用 j 作为索引遍历当前状态所有产生式
30                 if (index[j] != L[k].right[j].size()) { // 说明j不是规约项
31                     // 若规约项的Follow集合和移进项”点的位置“的”下一位“的First集合有交集，则不是SLR(1)文法
32                     if (judge2(First(L[k].right[j][index[j] + 1]), Follow(L[k].left[v[i]]))) {
33                         return false;
34                     }
35                 }
36             }
37         }
38     }
39 }
40 return true;
41 }

```

举例子理解师兄代码

举一个讲稿的例子，理解规约-移进冲突的代码：

文法：

```

1 S -> I|other
2 I -> if S| if S else S

```

假设当前是第 **k** 个状态：

```

1 I -> if S.
2 I -> if S. else S

```

当前状态下出现了规约-移进冲突，当前规约项的 Follow 集合就是 Follow(I)，当前移进项”点的位置“的”下一位“是 else，else 的 first 集合是 first(else)

手工做法：那么就求 `Follow(I)` 以及 `first(else)` 看看是否有交集。从文法中可以看出，`Follow(I) = { $, else }`，而对于终结符号 `else` 来说，`first(else) = { else }`，所以 `Follow(I) ∩ first(else) = { else }`，因此他们的交集不为空，故不是SLR(1)文法。

对这行代码的分析：

```
1 if (judge2(First(L[k].right[j][index[j] + 1]), Follow(L[k].left[v[i]])))
```

1. `First(L[k].right[j][index[j] + 1])`，`L[k]` 表示当前状态 `k`，`L[k].right[j]` 表示第 `j` 个产生式中右侧部分的 `string` 字符串（即 `if S else S`）

`right` 是一个字符串数组，用 `right[]` 返回数组里面的字符串，用 `right[][]` 返回这个字符串里的字符

2. `L[k].right[j][index[j] + 1]`，表示位于 `index[j]` 位置上的后一个字符（即 `else`）

3. `Follow(L[k].left[v[i]])` 返回当前状态 `k`，第 `i` 个规约项的"左侧部分"的 `Follow` 集合。

4. 最后再用 `judge2` 这个函数，判断两个集合是否有交集，有交集返回 `True`。

拓展

📌 实验四的求first和follow集合写代码一般放到第三题（属于自顶向下的内容，但实验是在自底向上中），代码也可以参照：1. [📖 手撕实验代码](#) 2. [📖 编译原理 复习提纲（浴帘👉）](#) 3. [📖 实验四考试代码\(by 郑伟楠\)](#)

实验四详细分析请见：[📖 实验四考试代码\(by 郑伟楠\)](#)

2. 题型二

四、LR 分析题（1 题，18 分）

1. 如果我们为教科书中的 TINY 语言增加书写格式类似于 C 语言的 for 循环语句。

那么请完成以下问题：【TINY 语言语法规则见后面附录】

- （1）请写出所添加语句对应的语法规则。
- （2）判断该文法是否为 LR(0) 文法。请说明原因。
- （3）判断该文法是否为 SLR(1) 文法。请说明原因。

四、LR 分析题（1 题，16 分）

1. 请写出支持选择（|）、连接、闭包（*）、可选（?）和括号等运算的正则表达式的文法，并画出该文法的 LR(0) DFA 图和判断其是否为 SLR(1) 文法。

默写文法 + 手撕LR(0)DFA图/SLR(1)分析表 + 判断是否符合文法

默写文法参考 [📖 编译原理 复习提纲](#)

手撕LR(0)DFA图/SLR(1)分析表 找视频学一学

判断是否符合语法规则：

LR0：有任何移进归约、归约归约冲突均不符合

SLR1：与LR0区别在于：

移进规约冲突必须这个终结符在FOLLOW(B)中且有B的规约项

归约归约冲突的交集一定要不为空

SLR(1) 文法 (SLR(1) grammar)

- 当且仅当对于任何状态 s ，以下的两个条件：
 - 1) 对于在 s 中的任何项目 $A \rightarrow \alpha.X\beta$ ，当 X 是一个终结符，且 X 在 $\text{Follow}(B)$ 中时， s 中没有完整的项目 $B \rightarrow \gamma.$ 。 [移进-归约冲突]
 - 2) 对于在 s 中的任何两个完整项目 $A \rightarrow \alpha.$ 和 $B \rightarrow \beta.$ ， $\text{Follow}(A) \cap \text{Follow}(B)$ 为空。 [归约-归约冲突]
- 均满足时，文法为 SLR(1) 文法。

五、语义分析题

1. 题型一

根据实际问题写出文法和语义动作

例题1:

- **问题1:** 如何将中缀表示转换成相应的后缀表示。
- **问题2:** 如何计算一个后缀表达式的值。

问题1参考解答:

PS: 只做简单的+*运算

规则

$E' \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow i$

语义函数

$\{ \}$

$\{ \text{Post}[p] = '+'; p = p + 1 \}$

$\{ \}$

$\{ \}$

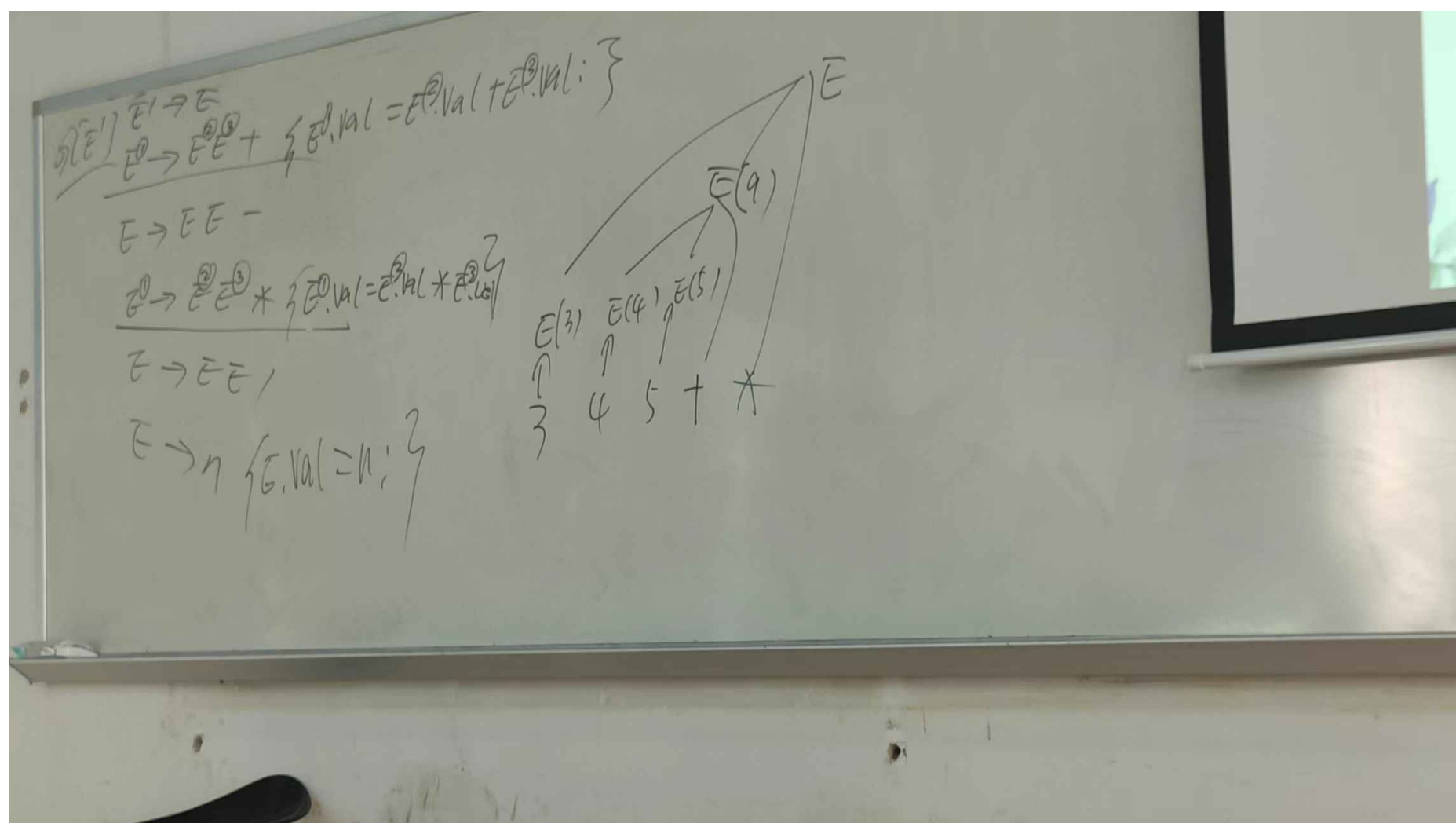
$\{ \text{Post}[p] = '*'; p = p + 1 \}$

$\{ \}$

$\{ \}$

$\{ \text{Post}[p] = i; p = p + 1 \}$

问题2参考解答：



同时此处回忆一下两个小测题：

小测一：

Q：实现算术表达式的文法

A：G[E]：

$$E \rightarrow E+T \mid E-T \mid T$$

$$T \rightarrow T * P \mid T / P \mid T \% P \mid P$$

$$P \rightarrow P \wedge F \mid F$$

$$F \rightarrow (E) \mid n$$

↓ 类图

小测二：

Q：实现正则表达式的文法（或、连接、闭包、基本）4层。

A：优先级：闭包 > 连接 > 或

$$\begin{array}{l} E \rightarrow E \mid T \mid T \\ T \rightarrow T P \mid P \\ P \rightarrow P^* \mid F \\ F \rightarrow (E) \mid c \end{array}$$

整理格式

G[R_E]：

$$R_E \rightarrow R_E " \mid " R_T \mid R_T$$

$$R_T \rightarrow R_T R_C \mid R_C$$

$$R_C \rightarrow R_C^* \mid R_F$$

$$R_F \rightarrow (R_E) \mid c$$

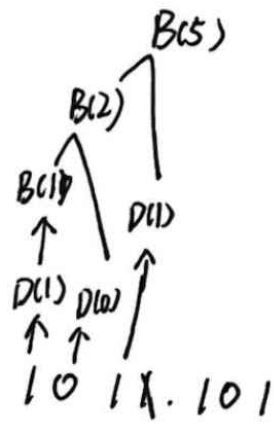
由小测一推导

例题2.1：写出二进制整数转十进制的文法及其语义动作（小测）

- 1 $B' \rightarrow B \{ \}$
- 2 $B \rightarrow BD \{ B1.val = B2.val * 2 + D.val \}$
- 3 $B \rightarrow D \{ B.val = D.val \}$
- 4 $D \rightarrow 0 \{ D.val = 0 \}$
- 5 $D \rightarrow 1 \{ D.val = 1 \}$

例题2.2 写出二进制转十进制浮点数的文法及其语义动作（小测）

带小数的二进制数转换为十进制数.



$$1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

~~1~~

~~1~~

$$\frac{1}{8} \times \frac{1}{2} + 0 \times \frac{1}{4} + 1 \times \frac{1}{8}$$

↓

$$\frac{1}{8} \times 2^2 + \frac{1}{8} \times 2^1 + \frac{1}{2} \times 2^0$$

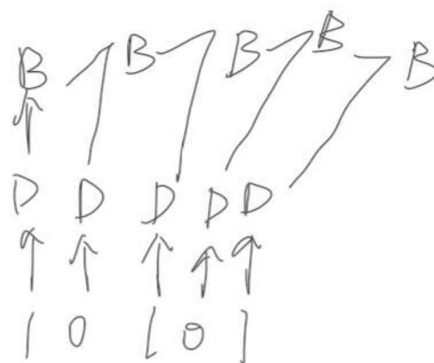
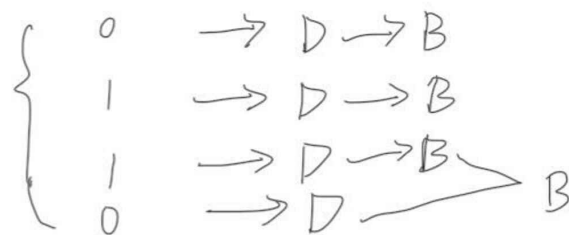
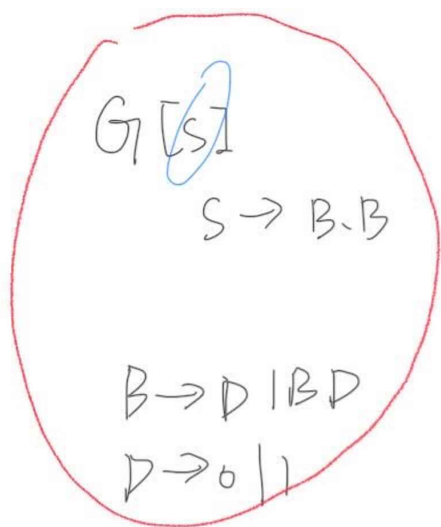
$$G[B'] \quad B' \rightarrow B^0 B^1 \quad B'.val = B^0.val + B^1.val / power(2, B^0.len);$$

$$B \rightarrow D \quad \{ B.val = D.val \times 2 \quad B.len = D.len;$$

$$B^0 \rightarrow B^1 D \quad \{ B^0.val = B^0.val \times 2 + D.val, B^0.len = B^0.len + D.len;$$

$$D \rightarrow 0 \quad \{ D.val = 0, D.len = 1;$$

$$D \rightarrow 1 \quad \{ D.val = 1, D.len = 1;$$



从基字符开始表达。

可以0开头，不以0结束。

编号要引回。

转为 + (括号) :

左递归?
 $S.val = B^0.val + B^0.val / pow(2, B^0.len)$

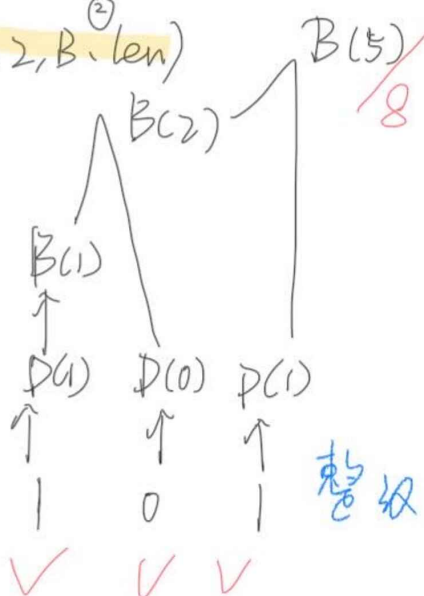
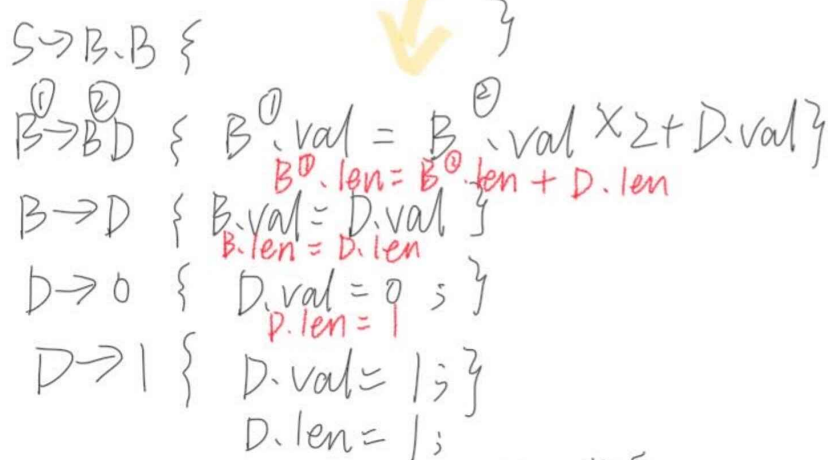
Tips:

通过 len 的

控制解决了

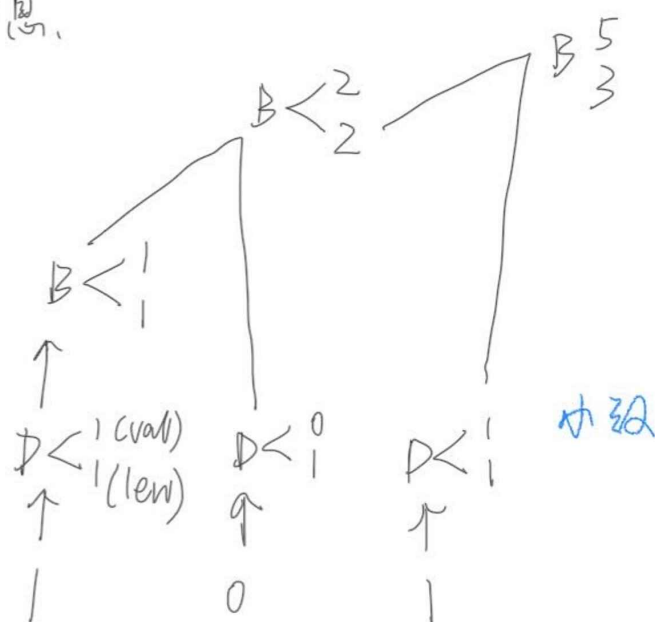
整数、小数计算

的冲突。



补充信息。

$\frac{5}{8} = \frac{5}{2^3} = \frac{5}{2^{(length)}}$
 要记录长度



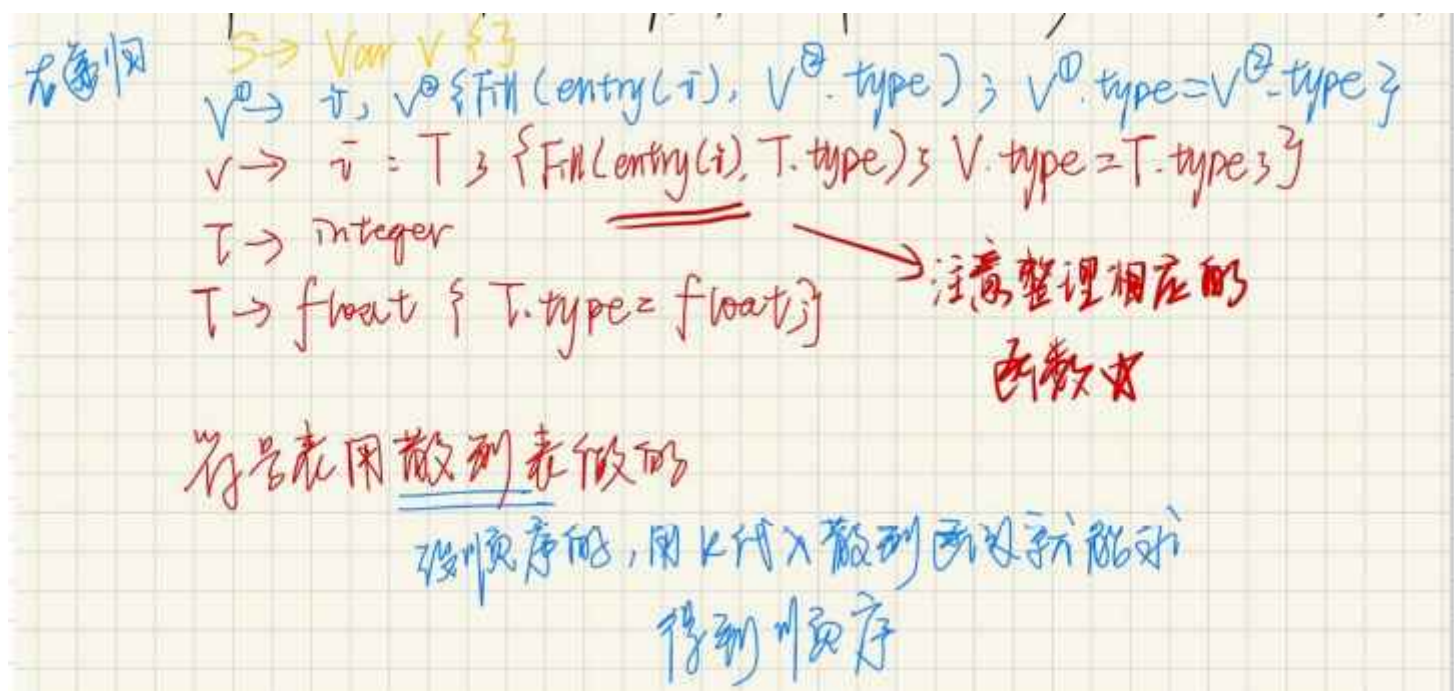
例题3:

写出文法和语义动作

```

1 var i:integer;
2 var i,j,k:float;

```



顺带整理一下对应的常用函数：

```

1 Pointer ENTRY(NAME)
2 {
3     ENTRYNO = LOOKUP(NAME);
4     if(ENTRYNO == NULL) return(ENTER(NAME));
5 }

```



LOOKUP(target)：在已有的表中找一个target

ENTER(target)：将target添加入表中

ENTRY(target)：获得target索引的目标（不存在就新增，存在就直接返回）

2. 题型二

将代码转成用四元组表达的中间代码

例题：请将下面代码转为四元组的中间代码

```

1  if(A<B&&C>D || E!=F)
2      if(X>Y&&S<T) G=0; else G=1;
3  else H=1;

```

参考答案：

```

1  (J<,A,B,3)
2  (J,,,5)
3  (J>,C,D,7)
4  (J,,,5)
5  (J!=,E,F,7)
6  (J,,,15)
7  (J>,X,Y,9)
8  (J,,,13)
9  (J<,S,T,11)
10 (J,,,13)

```



```
11 (=,0,,G)
12 (J,,,16)
13 (=,1,,G)
14 (J,,,16)
15 (=,1,,H)
16
```

六、综合分析设计题

1. 题型一（抽象题）

当有多个矩阵进行运算时，各个矩阵之间的运算顺序可影响到最终的运算次数。如 A 为 50×20 的矩阵，B 为 20×5 的矩阵，C 为 5×30 的矩阵，那么 $(AB)C$ 的运算次数为 12500 次， $A(BC)$ 的运算次数为 33000 次，下面请你用编译原理的方法完成这一类式子的计算和判别功能，写出相关思路 and 代码，要求的输入输出格式如下：↵

```
Input:↓
(ABC↓
(AB)C↓
A(BC)↓
Output:↓
No↓
12500↓
33000↵
```

不用编译原理方法最多得到 3 分（估计用算法设计与分析的方法的话有三分，写对文字的话最多 2 分）↵

下面这个是我的解法感觉应该可以拿到一点分？（太抽象了这个题目）（by:邹锐城）

文法

$$S \rightarrow TCV)T$$

$$V \rightarrow Va|a|(Va)$$

$$T \rightarrow Ta|a$$

↓
①为矩阵连乘 交给V
把行5]

的括号是((,))的, 不是(,), (,), (,)的

$$V^1 \rightarrow V^2 a \begin{cases} V^1.count = V^2.row \times V^2.col \times a.col \\ V^1.row = V^2.row \\ V^1.col = a.col \end{cases}$$

带括号也一样

$$S \rightarrow \underbrace{V^1(V^2)V^3} \begin{cases} S.count = V^1.count + V^2.count + V^3.count \\ S.col = V^1.col \\ S.row = V^1.row \end{cases}$$

三个矩阵连乘相当于

$$S.count += V^1.row \times V^1.col \times V^2.col + V^2.row \times V^2.col \times V^3.col;$$

递归下降解法，输入作了简单处理(by:YJ)

文法

- 1 $E \rightarrow E * T \mid T$
- 2 $T \rightarrow (E) \mid a$
- 3 a表示矩阵
- 4 写成EBNF
- 5 $E \rightarrow T \{ * T \}$
- 6 $T \rightarrow (E) \mid a$

语义动作

- 1 $E \rightarrow E' * T \{$
- 2 $E.cnt = E'.cnt + E'.row * E'.col * T.col + T.cnt;$
- 3 $E.row = E'.row;$
- 4 $E.col = T.col;$
- 5 $\}$
- 6
- 7 $E \rightarrow T \{$
- 8 $E.cnt = T.cnt;$

```

9      E.row = T.row;
10     E.col = T.col;
11 }
12
13 T-> (E) {
14     T.cnt = E.cnt;
15     T.row = E.row;
16     T.col = E.col;
17 }
18
19 T-> a {
20     T.cnt = 0;
21     T.row = a.row;
22     T.col = a.col;
23 }

```

程序

这里仅作为练习，输入简化为：10.20 表示一个10行20列的矩阵，加入*表示矩阵相乘。

没有加入报错机制，真要写的时候自己改一下就好。

```

1  struct Matrix {
2      int row;
3      int col;
4      int cnt;
5      Matrix(int _row = 0, int _col = 0) : row(_row), col(_col) {
6          cnt = 0;
7      }
8  };
9
10 char str[110];
11 int p, len;
12
13 Matrix E();
14 Matrix T();
15
16 // E-> E*T | T
17 // T-> (E) | a
18
19 // E-> T {* T}
20 // T-> (E) | a
21
22 Matrix E() {
23     Matrix m = T();
24     while (p < len && str[p] == '*') {
25         ++p;
26         Matrix rhs = T();
27         m.cnt += m.row * m.col * rhs.col + rhs.cnt;
28         m.col = rhs.col;
29     }
30     return m;
31 }
32
33 Matrix T() {
34     Matrix m;
35     if (str[p] == '(') {
36         ++p;
37         m = E();
38         ++p;
39     } else if (isdigit(str[p])) {







```

```

40     while (isdigit(str[p])) {
41         m.row = m.row * 10 + str[p++] - '0';
42     }
43     ++p;
44     while (isdigit(str[p])) {
45         m.col = m.col * 10 + str[p++] - '0';
46     }
47 }
48 return m;
49 }
50
51 int main() {
52     std::cin >> str;
53     len = strlen(str);
54     p = 0;
55     Matrix m = E();
56     std::cout << m.row << ' ' << m.col << ' ' << m.cnt;
57     return 0;
58 }

```

输入输出

Input:	Copy
50.10*(10.20*20.50)	
Expected Output:	Copy
Received Output:	Copy
50 50 35000	
^ TC 2 Failed 579ms	 
Input:	Copy
(50.10*10.20)*20.50	
Expected Output:	Copy
Received Output:	Copy
50 50 60000	
^ TC 3 Failed 568ms	 
Input:	Copy
(50.20*20.5)*5.30	
Expected Output:	Copy
Received Output:	Copy
50 30 12500	
^ TC 4 Failed 629ms	 
Input:	Copy
50.20*(20.5*5.30)	
Expected Output:	Copy
Received Output:	Copy
50 30 33000	

2. 题型2 (TINY扩充)

1. 我们知道，原来的TINY语言只支持十进制整型数的处理。如果我们现在要对tiny语言进行扩充，让其能支持处理十进制浮点数（包括浮点数变量和浮点数常量，而浮点数常数又包含有带小数部分和科学记数两种表示方法）。现请你使用编译原理中有关的知识与原理来解决这个问题。【TINY语言文法规则见后面附录】

（1）文法规则需要做哪些改写，并写出改写后的文法规则。

（2）根据问题解决时的实际需要，写出需要对原tiny语言的词法分析程序、语法分析程序以及语义分析程序做出怎样的修改。

↩

6. 对教材后面的TINY语言（这个百度也有文法规则）进行改写，实现一个类似于C语言for语句的表达式，并且写出语法树的生成程序（递归下降加语义动作）↩

↩

1. 在实验三中，我们为教科书中的TINY语言进行了语法的扩充。我们知道，该TINY语言不支持类似于C语言的数据类型定义语句。现想为其做扩充，让其能支持这语句。请你根据实验三的实践经验，完成以下内容：【TINY语言文法规则见后面附录】

（1）文法规则该如何改写扩充。【说明：支持数据类型为int, bool, char, real】

（2）根据实际的需要，该如何改写词法、语法的分析程序，请简明扼要地描述你的做法。

（3）根据实际的需要，写出该扩充文法的语义分析程序。

第一问，如何改写扩充，其实还是写文法，自求多福

第二问，如何改写词法、语法程序（或语义分析程序），应该存在一个较为通用的答案，如：

词法程序：在 `global.h` 中加入对应的TokenType，在 `scan.cpp` 中的 `reservedWords[MAXRESERVED]` 结构体数组加入相应的关键字，并在 `getToken()` 加入对应的case。

语法程序：在 `parse.cpp` 中先加入对应的函数声明，再通过文法写出对应的函数

语义分析（不确定）：在 `parse.cpp` 中的对应语法树生成函数里面写个 `GEN()` 函数（生成四元组的）或者就在对应地方写语义函数

3. 题型三：设计文法生成语法树

参见 [国编译原理复习提纲](#)（浴帘🧘）