编译原理 实验二：语法分析

09019106 牟倪

目录

一、实验目标

- 理解语法分析过程。
- 对 LL parsing，熟练掌握消除左递归→消除左公共因子→得到 First 和 Follow 集合→构造 parsing table 的过程。
- 对 LR parsing，熟练掌握构造 NFA→化为 DFA→构造 parsing table 的过程。
- 自主实现 LL 或 LR 语法分析程序。

二、实验内容

- 自主定义 token 和 context-free grammar。
- 完成词法分析程序：
  - 输入 token 串，输出解析结果的 parsing tree 的前序和后序遍历。
  - 对不合法的输入 token 串，报错。

三、基本思路

我选择实现 LL 分析程序。首先对原 grammar 消除左递归和左公共因子，然后得到 First 和 Follow 集合，最后构造 parsing table。将 token 序列成功解析后，我们用树来记录 parsing tree 的信息，并输出 parsing tree 的遍历序列。

四、假设与名词定义

终结符定义：

- id：表示变量的 identifier；
- =：赋值的等于号。
- num：表示数字。
- if：表示“如果”，应用格式为 if then else。
- then：表示“然后”，应用格式为 if then else。
- else：表示“其他”，应用格式为 if then else。
- cmp：表示比较的运算符，如 <、>、≤、≥ 等。
- op：运算符号，如 +、-、*、/ 等。
- rop：逻辑运算符号，如 &&、|| 等。

非终结符定义：

- S：文法开始符号，表示 statement。

- E：表示 expression，返回值为数值。
- B：表示 boolean expression，返回 true/false。

context-free grammar 定义：

- S → if B then S else S
- S → id = E
- B → E cmp E
- B → E cmp E rop B
- E → num
- E → num op E

五、构造 Parsing Table

5.1 消除左递归与左公共因子

grammar 中没有左递归，但 B 和 E 的产生式有左公共因子。

把所有的 E 替换为 num Enext，

- Enext → op num Enext
- Enext → ε

消除 E 产生式中的左公共因子后，文法如下所示：

- S → if B then S else S
- S → id = num Enext
- B → num Enext cmp num Enext
- B → num Enext cmp num Enext rop B
- Enext → op num Enext
- Enext → ε

把所有的 B 替换为 num Enext cmp num Enext Bnext，

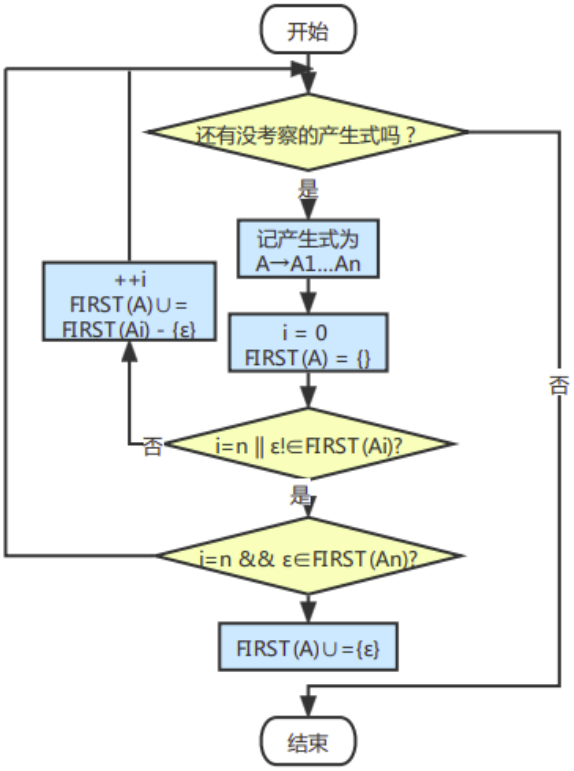
- Bnext → rop num Enext cmp num Enext Bnext
- Bnext → ε

消除 B 产生式中的左公共因子后，文法如下所示：

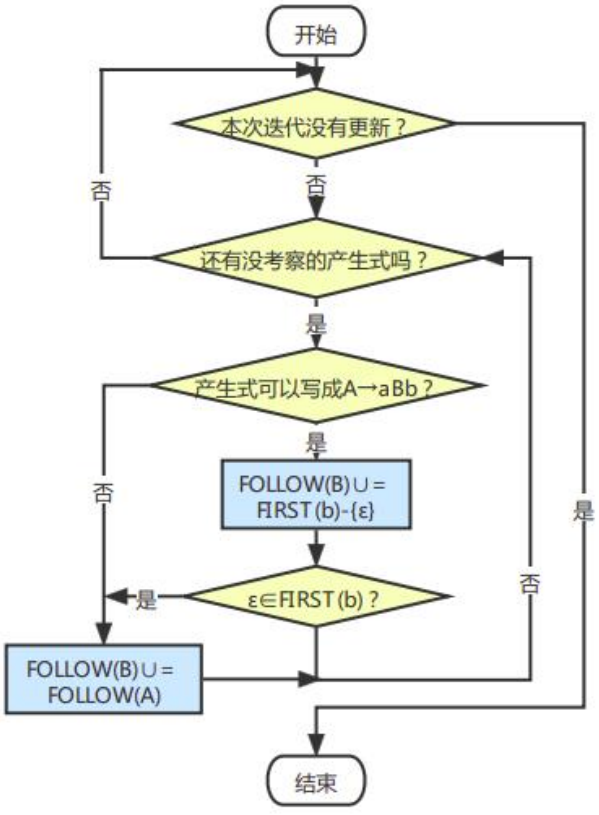
- S → if num Enext cmp num Enext Bnext then S else S
- S → id = num Enext
- Bnext → rop num Enext cmp num Enext Bnext
- Bnext → ε
- Enext → op num Enext
- Enext → ε

5.2 计算 First 与 Follow 集合

构造 First 集的算法如下所示：



构造 Follow 集的算法如下所示：

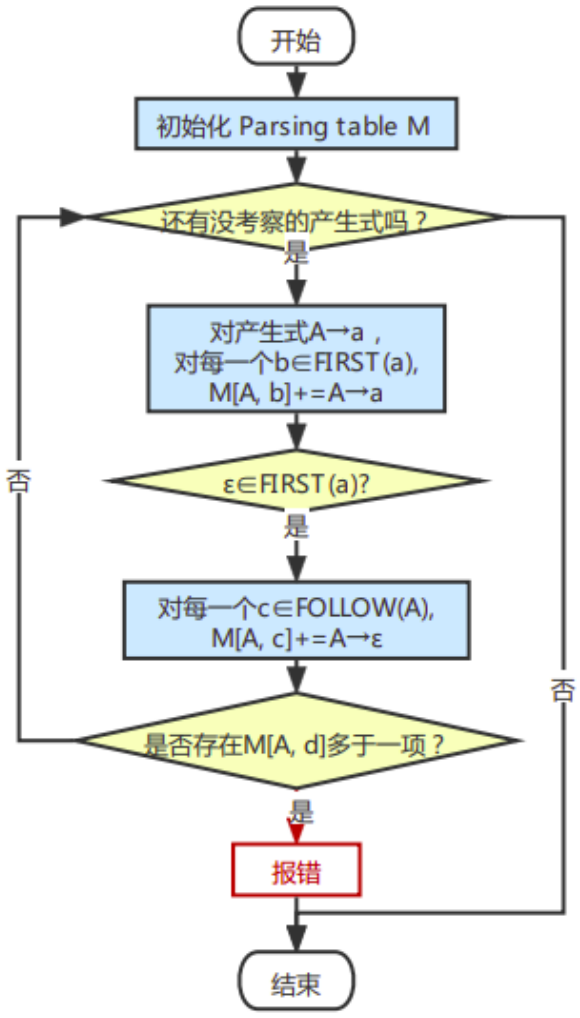


目前，非终结符有：S, Bnext, Enext；终结符有：if, then, else, id, =, num, cmp, rop, op。我们构造 First 集和 Follow 集如下：

		FISRT	FOLLOW
S	S → if num Enext cmp num Enext Bnext then S else S	{if}	{else, #}
	S → id = num Enext	{id}	
Bnext	Bnext → rop num Enext cmp num Enext Bnext	{rop}	{then}
	Bnext → ε	{ε}	
Enext	Enext → op num Enext	{op}	{cmp, then, rop, else, #}
	Enext → ε	{ε}	

5.3 构造 Parsing Table

构造 parsing table 的算法如下所示：



构造 parsing table 如下：

	if	then	else	id	=	num	cmp	rop	op	#
S	S → if num Enext cmp num Enext Bnext then S else S			S → id = num Enext						
Bnext		Bnext → ε						Bnext → rop num Enext cmp num Enext Bnext		
Enext		Enext → ε	Enext → ε				Enext → ε	Enext → ε	Enext → op num Enext	Enext → ε

六、核心数据结构

6.1 用栈实现自顶而下分析

在最初，栈底为语法开始符号 S。当读入 token 后，如果栈顶符号位终结符（即 token），比较两个 token 是否相等，若相等则弹栈、继续读入 token，若不相等则报错。如果栈顶符号为非终结符，则根据 parsing table 进行推导，即将产生式左部从栈中弹出，然后将产生式右部压入栈中，读入 token 的进度不变化。

6.2 用树记录 parsing tree 信息

我们构造节点类 `Node`，包含：

- `string value`，代表节点存储的值。如果是叶节点，`value` 为终结符；如果是非叶节点，`value` 为非终结符。
- `vector<Node*> children`，代表指向子节点的指针。

每完成一次推导，我们就把弹出栈的产生式左部（即非终结符）拿到，并相应去构造产生式右部的节点，令产生式左部的节点的 `children` 指向产生式右部的节点，并将这些产生式右部节点压栈。

七、核心算法

7.1 用 map 记录产生式

我们将 token 以 string 的形式记录，用 map<int, vector<string>>的形式记录产生式，其中 map 的 key 为整型，是约定好的产生式编号。代码如下：

```
map<int, vector<string>> productions;

void init(){
    // productions
    string temp1[11] = {"if", "num", "Enext", "cmp", "num", "Enext", "Bnext", "then", "S", "else", "S"};
    productions[1] = vector<string>(temp1, temp1 + 11);

    string temp2[4] = {"id", "=", "num", "Enext"};
    productions[2] = vector<string>(temp2, temp2 + 4);

    string temp3[7] = {"rop", "num", "Enext", "cmp", "num", "Enext", "Bnext"};
    productions[3] = vector<string>(temp3, temp3 + 7);

    productions[4] = vector<string>();

    string temp5[3] = {"op", "num", "Enext"};
    productions[5] = vector<string>(temp5, temp5 + 3);

    productions[6] = vector<string>();
}
```

7.2 硬编码构造 parsing table

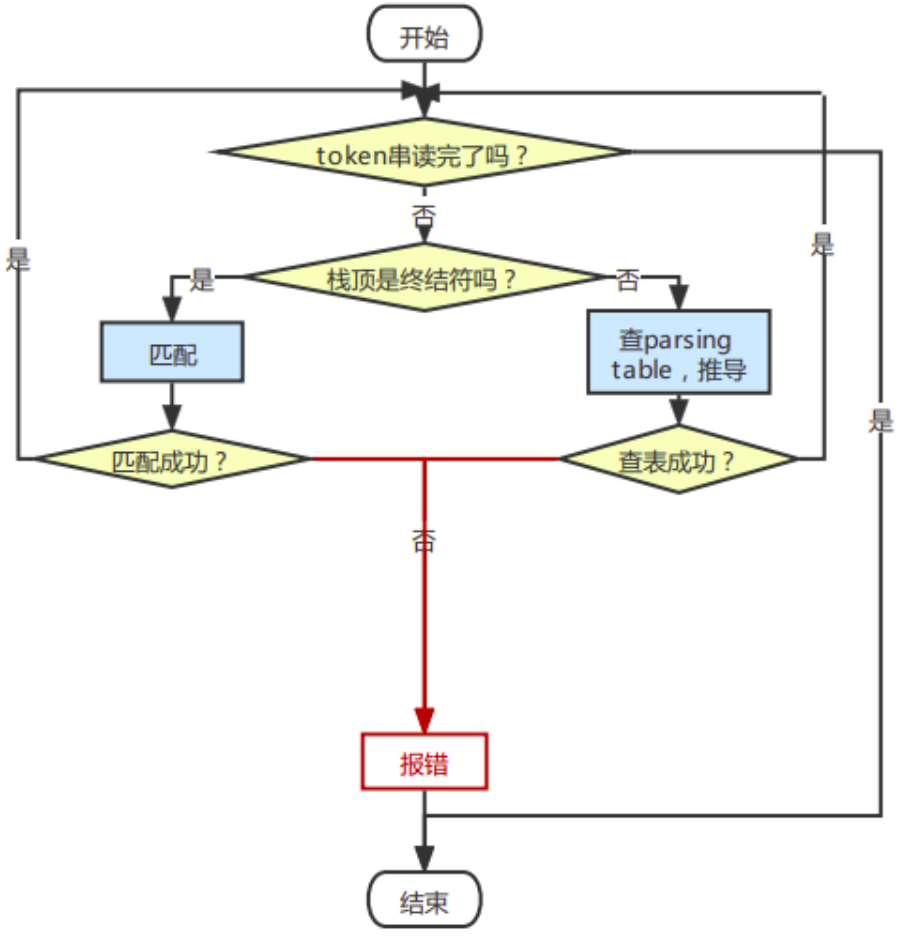
我们用函数 int parsing_table(string stacktop, string nextval)来完成查 parsing table 的动作，参数 1 为栈顶 token，参数 2 为下一个 token，返回要使用的产生式编号。代码如下：

```
int parsing_table(string stacktop, string nextval){
    if(stacktop == "S"){
        if(nextval == "if") return 1;
        if(nextval == "id") return 2;
        fail();
```

```
    }
    else if(stacktop == "Bnext"){
        if(nextval == "rop") return 3;
        if(nextval == "then") return 4;
        fail();
    }
    else if(stacktop == "Enext"){
        if(nextval == "op") return 5;
        if(nextval == "then" || nextval == "else" ||
            nextval == "cmp" || nextval == "rop" || nextval == "#
    ") return 6;
        fail();
    }
    return -1;
}
```

7.3 解析 token 串并构造 parsing tree

我们用函数 `void parse(vector<string> tokens)`来完成 token 串的解析。该函数会修改作为全局变量的 parsing tree 数据结构。流程图如下：



代码如下：

```
void parse(vector<string> tokens){
    root = new Node("S");
    st.push(root);
    int cur = 0;
    while(!st.empty() && tokens[cur] != "#"){
        string stacktop = st.top()->value;
        if(!is_non_terminal(stacktop)){ // terminal
            if(stacktop == tokens[cur]){
                st.pop();++cur;
            }
            else fail();
        }
        else{ // non terminal
            string nextval = tokens[cur];
            int idx = parsing_table(stacktop, nextval);
            Node *temp = st.top();st.pop();
            int len = productions[idx].size();
            for(int i=0; i<len; ++i){
                Node* newNode = new Node(productions[idx][i]);
                temp->children.push_back(newNode);
            }
            for(int i=len-1; i>=0; --i){
                st.push(temp->children[i]);
            }
        }
    }
}
```

7.4 parsing tree 的前序和后序遍历

我们用递归的方式实现 parsing tree 的前序和后序遍历。代码如下：

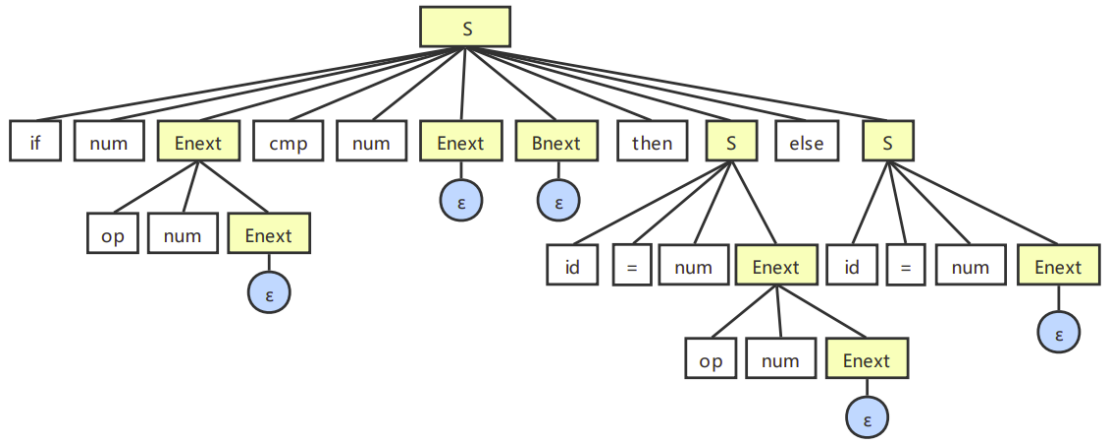
```
void preorder_traversal(Node* node){
    cout<<node->value<<" ";
    for(Node* i:node->children) preorder_traversal(i);
}

void postorder_traversal(Node* node){
    for(Node* i:node->children) postorder_traversal(i);
    cout<<node->value<<" ";
}
```

八、实验

测试用例为 "if num op num cmp num then id = num op num else id = num #"。这样看可能比较抽象，我们将其实例化一下：

```
1. if a + b > c then x = a + b else x = c
```

这就是一个以上 token 序列的实例化。parsing tree 如下所示，其中黄色节点为非终结符，白色节点为终结符。



其前序遍历序列为：S if num Enext op num Enext cmp num Enext Bnext then S id = num Enext op num Enext else S id = num Enext

后序遍历序列为：if num op num Enext Enext cmp num Enext Bnext then id = num op num Enext Enext S else id = num Enext S S

程序执行结果：

```
preorder:
S if num Enext op num Enext cmp num Enext Bnext then S id = num Enext op num Enext else S id = num Enext

postorder:
if num op num Enext Enext cmp num Enext Bnext then id = num op num Enext Enext S else id = num Enext S S
```

前序遍历与后序遍历都输出正确。

我们再来测试一个失败的用例 "if then else #"，程序报错：

```
parsing failed!
```

可以看出，实验结果正确，程序编写合理。

九、出现问题与解决方案

没有出现问题。

十、实验体会

经过本次实验，我复习了自顶而下 LL 语法分析的内容，消除左递归、左公共因子、计算 First 和 Follow 集合、构造 parsing table 也更熟练了。

附录：源代码

```
#include<iostream>
#include<stack>
#include<vector>
#include<map>
#include<string>
#include<cstring>

using namespace std;

class Node{
public:
    string value;
    vector<Node*> children;
    Node(string v):value(v){}
    Node(){}
};

map<int, vector<string>> productions;
Node* root = NULL;
stack<Node*> st;

void fail(){
    cout<<"parsing failed!\n";
    exit(1);
}

void init(){
    // productions
    string temp1[11] = {"if", "num", "Enext", "cmp", "num", "Enext", "Bnext", "then", "S", "else", "S"};
    productions[1] = vector<string>(temp1, temp1 + 11);

    string temp2[4] = {"id", "=", "num", "Enext"};
    productions[2] = vector<string>(temp2, temp2 + 4);

```

```
35.     string temp3[7] = {"rop", "num", "Enext", "cmp", "num", "Enex
   t", "Bnext"};
36.     productions[3] = vector<string>(temp3, temp3 + 7);
37.
38.     productions[4] = vector<string>();
39.
40.     string temp5[3] = {"op", "num", "Enext"};
41.     productions[5] = vector<string>(temp5, temp5 + 3);
42.
43.     productions[6] = vector<string>();
44. }
45.
46. bool is_non_terminal(string token){
47.     return (token == "S" || token == "Bnext" || token == "Enext")
   ;
48. }
49.
50. int parsing_table(string stacktop, string nextval){
51.     if(stacktop == "S"){
52.         if(nextval == "if") return 1;
53.         if(nextval == "id") return 2;
54.         fail();
55.     }
56.     else if(stacktop == "Bnext"){
57.         if(nextval == "rop") return 3;
58.         if(nextval == "then") return 4;
59.         fail();
60.     }
61.     else if(stacktop == "Enext"){
62.         if(nextval == "op") return 5;
63.         if(nextval == "then" || nextval == "else" ||
64.             nextval == "cmp" || nextval == "rop" || nextval == "#
   ") return 6;
65.         fail();
66.     }
67.     return -1;
68. }
69.
70. void parse(vector<string> tokens){
71.     root = new Node("S");
72.     st.push(root);
73.     int cur = 0;
74.     while(!st.empty() && tokens[cur] != "#"){
75.         string stacktop = st.top()->value;
```

```
        if(!is_non_terminal(stacktop)){ // terminal
            if(stacktop == tokens[cur]){
                st.pop();++cur;
            }
            else fail();
        }
        else{ // non terminal
            string nextval = tokens[cur];
            int idx = parsing_table(stacktop, nextval);
            Node *temp = st.top();st.pop();
            int len = productions[idx].size();
            for(int i=0; i<len; ++i){
                Node* newNode = new Node(productions[idx][i]);
                temp->children.push_back(newNode);
            }
            for(int i=len-1; i>=0; --i){
                st.push(temp->children[i]);
            }
        }
    }

}

void preorder_traversal(Node* node){
    cout<<node->value<<" ";
    for(Node* i:node->children) preorder_traversal(i);
}

void postorder_traversal(Node* node){
    for(Node* i:node->children) postorder_traversal(i);
    cout<<node->value<<" ";
}

int main(){
    init();
    string temptoken[17] = {"if", "num", "op", "num", "cmp", "num",
        "then", "id", "=", "num", "op", "num",
        "else", "id", "=", "num", "#"
    };
    vector<string> token_to_parse = vector<string>(temptoken, temptoken + 17);
    //string badtoken[4] = {"if", "then", "else", "#"}; // 不合法的token序列，报错
```

```
    //vector<string> token_to_parse = vector<string>(badtoken, badtoken + 4);

    parse(token_to_parse);
    cout<<"preorder:\n";
    preorder_traversal(root); cout<<"\n\npostorder:\n";
    postorder_traversal(root); cout<<"\n\n";
}
```