编译原理 实验二：语法分析

09019106 牟倪

目录

[一、实验目标 2](#_Toc90831602)

[二、实验内容 2](#_Toc90831603)

[三、基本思路 2](#_Toc90831604)

[四、假设与名词定义 2](#_Toc90831605)

[五、构造Parsing Table 3](#_Toc90831606)

[5.1 消除左递归与左公共因子 3](#_Toc90831607)

[5.2 计算First与Follow集合 4](#_Toc90831608)

[5.3 构造Parsing Table 5](#_Toc90831609)

[六、核心数据结构 6](#_Toc90831610)

[6.1 用栈实现自顶而下分析 6](#_Toc90831611)

[6.2 用树记录parsing tree信息 6](#_Toc90831612)

[七、核心算法 7](#_Toc90831613)

[7.1 用map记录产生式 7](#_Toc90831614)

[7.2 硬编码构造parsing table 7](#_Toc90831615)

[7.3解析token串并构造parsing tree 8](#_Toc90831616)

[7.4 parsing tree的前序和后序遍历 10](#_Toc90831617)

[八、实验 10](#_Toc90831618)

[九、出现问题与解决方案 11](#_Toc90831619)

[十、实验体会 11](#_Toc90831620)

[附录：源代码 12](#_Toc90831621)

# 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，

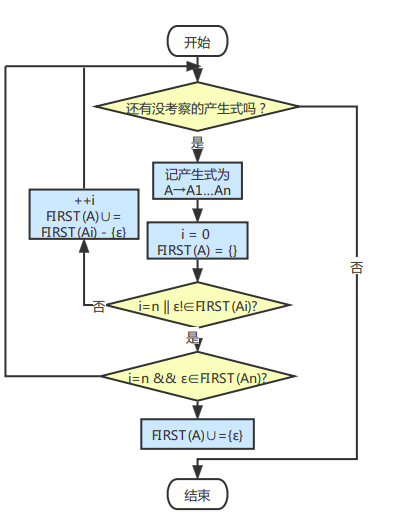
* Bnext → rop num Enext cmp num Enext Bnext
* Bnext → ε

消除B产生式中的左公共因子后，文法如下所示：

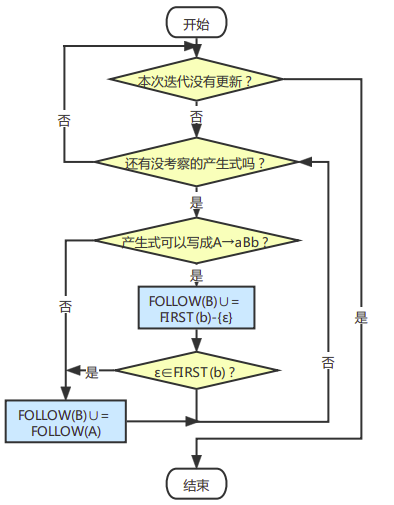
* S → if num Enext cmp num Enext Bnext then S else S
* S → id = num Enext
* Bnext → rop num Enext cmp num Enext Bnext
* Bnext → ε
* Enext → op num Enext
* Enext → ε

## 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：

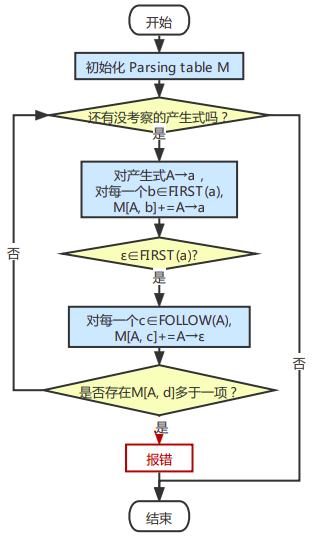


目前，非终结符有：S, Bnext, Enext；终结符有：if, then, else, id, =, num, cmp, rop, op。我们构造First集和Follow集如下：

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 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：

1. map<int, vector<string>> productions;
2. void init(){
3. *// productions*
4. string temp1[11] = {"if", "num", "Enext", "cmp", "num", "Enext", "Bnext", "then", "S", "else", "S"};
5. productions[1] = vector<string>(temp1, temp1 + 11);
6. string temp2[4] = {"id", "=", "num", "Enext"};
7. productions[2] = vector<string>(temp2, temp2 + 4);
8. string temp3[7] = {"rop", "num", "Enext", "cmp", "num", "Enext", "Bnext"};
9. productions[3] = vector<string>(temp3, temp3 + 7);
10. productions[4] = vector<string>();
11. string temp5[3] = {"op", "num", "Enext"};
12. productions[5] = vector<string>(temp5, temp5 + 3);
13. productions[6] = vector<string>();
14. }

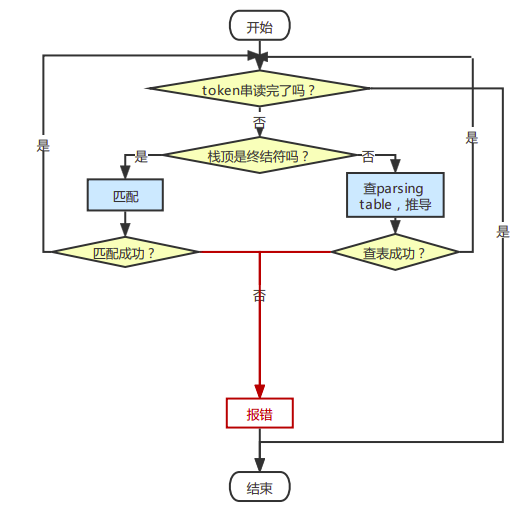
## 7.2 硬编码构造parsing table

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：

1. int parsing\_table(string stacktop, string nextval){
2. if(stacktop == "S"){
3. if(nextval == "if") return 1;
4. if(nextval == "id") return 2;
5. fail();
6. }
7. else if(stacktop == "Bnext"){
8. if(nextval == "rop") return 3;
9. if(nextval == "then") return 4;
10. fail();
11. }
12. else if(stacktop == "Enext"){
13. if(nextval == "op") return 5;
14. if(nextval == "then" || nextval == "else" ||
15. nextval == "cmp" || nextval == "rop" || nextval == "#") return 6;
16. fail();
17. }
18. return -1;
19. }

## 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：

1. void parse(vector<string> tokens){
2. root = new Node("S");
3. st.push(root);
4. int cur = 0;
5. while(!st.empty() && tokens[cur] != "#"){
6. string stacktop = st.top()->value;
7. if(!is\_non\_terminal(stacktop)){ *// terminal*
8. if(stacktop == tokens[cur]){
9. st.pop();++cur;
10. }
11. else fail();
12. }
13. else{ *// non terminal*
14. string nextval = tokens[cur];
15. int idx = parsing\_table(stacktop, nextval);
16. Node \*temp = st.top();st.pop();
17. int len = productions[idx].size();
18. for(int i=0; i<len; ++i){
19. Node\* newNode = new Node(productions[idx][i]);
20. temp->children.push\_back(newNode);
21. }
22. for(int i=len-1; i>=0; --i){
23. st.push(temp->children[i]);
24. }
25. }
26. }
27. }

## 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：

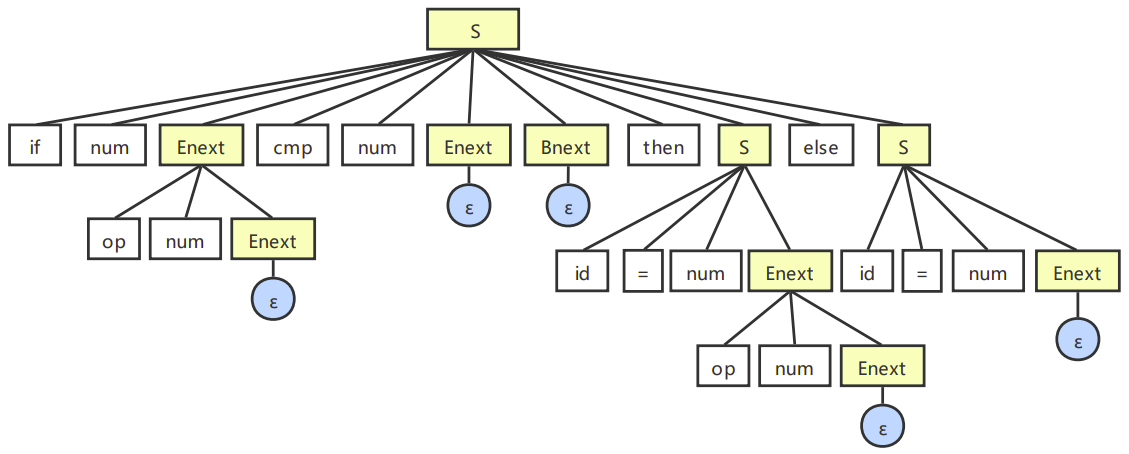
1. void preorder\_traversal(Node\* node){
2. cout<<node->value<<" ";
3. for(Node\* i:node->children) preorder\_traversal(i);
4. }
5. void postorder\_traversal(Node\* node){
6. for(Node\* i:node->children) postorder\_traversal(i);
7. cout<<node->value<<" ";
8. }

# 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

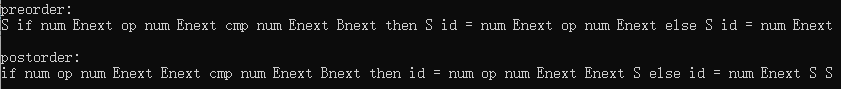
这就是一个以上token序列的实例化。parsing tree如下所示，其中黄色节点为非终结符，白色节点为终结符。



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：



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：

E:\temp\1639909459(1).png

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

1. #include<iostream>
2. #include<stack>
3. #include<vector>
4. #include<map>
5. #include<string>
6. #include<cstring>
7. using namespace std;
8. class Node{
9. public:
10. string value;
11. vector<Node\*> children;
12. Node(string v):value(v){}
13. Node(){}
14. };
15. map<int, vector<string>> productions;
16. Node\* root = NULL;
17. stack<Node\*> st;
18. void fail(){
19. cout<<"parsing failed!\n";
20. exit(1);
21. }
22. void init(){
23. *// productions*
24. string temp1[11] = {"if", "num", "Enext", "cmp", "num", "Enext", "Bnext", "then", "S", "else", "S"};
25. productions[1] = vector<string>(temp1, temp1 + 11);
26. string temp2[4] = {"id", "=", "num", "Enext"};
27. productions[2] = vector<string>(temp2, temp2 + 4);
28. string temp3[7] = {"rop", "num", "Enext", "cmp", "num", "Enext", "Bnext"};
29. productions[3] = vector<string>(temp3, temp3 + 7);
30. productions[4] = vector<string>();
31. string temp5[3] = {"op", "num", "Enext"};
32. productions[5] = vector<string>(temp5, temp5 + 3);
33. productions[6] = vector<string>();
34. }
35. bool is\_non\_terminal(string token){
36. return (token == "S" || token == "Bnext" || token == "Enext");
37. }
38. int parsing\_table(string stacktop, string nextval){
39. if(stacktop == "S"){
40. if(nextval == "if") return 1;
41. if(nextval == "id") return 2;
42. fail();
43. }
44. else if(stacktop == "Bnext"){
45. if(nextval == "rop") return 3;
46. if(nextval == "then") return 4;
47. fail();
48. }
49. else if(stacktop == "Enext"){
50. if(nextval == "op") return 5;
51. if(nextval == "then" || nextval == "else" ||
52. nextval == "cmp" || nextval == "rop" || nextval == "#") return 6;
53. fail();
54. }
55. return -1;
56. }
57. void parse(vector<string> tokens){
58. root = new Node("S");
59. st.push(root);
60. int cur = 0;
61. while(!st.empty() && tokens[cur] != "#"){
62. string stacktop = st.top()->value;
63. if(!is\_non\_terminal(stacktop)){ *// terminal*
64. if(stacktop == tokens[cur]){
65. st.pop();++cur;
66. }
67. else fail();
68. }
69. else{ *// non terminal*
70. string nextval = tokens[cur];
71. int idx = parsing\_table(stacktop, nextval);
72. Node \*temp = st.top();st.pop();
73. int len = productions[idx].size();
74. for(int i=0; i<len; ++i){
75. Node\* newNode = new Node(productions[idx][i]);
76. temp->children.push\_back(newNode);
77. }
78. for(int i=len-1; i>=0; --i){
79. st.push(temp->children[i]);
80. }
81. }
82. }
84. }
85. void preorder\_traversal(Node\* node){
86. cout<<node->value<<" ";
87. for(Node\* i:node->children) preorder\_traversal(i);
88. }
89. void postorder\_traversal(Node\* node){
90. for(Node\* i:node->children) postorder\_traversal(i);
91. cout<<node->value<<" ";
92. }
93. int main(){
94. init();
95. string temptoken[17] = {"if", "num", "op", "num", "cmp", "num",
96. "then", "id", "=", "num", "op", "num",
97. "else", "id", "=", "num", "#"
98. };
99. vector<string> token\_to\_parse = vector<string>(temptoken, temptoken + 17);
100. *//string badtoken[4] = {"if", "then", "else", "#"}; // 不合法的token序列，报错*
101. *//vector<string> token\_to\_parse = vector<string>(badtoken, badtoken + 4);*
102. parse(token\_to\_parse);
103. cout<<"preorder:\n";
104. preorder\_traversal(root); cout<<"\n\npostorder:\n";
105. postorder\_traversal(root); cout<<"\n\n";
106. }