**《编译原理》实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **2017级计算机科学与技术05班** | | **姓名** | **龙俊桦** |
| **实验题目** | 语法分析程序的设计与实现 | | | | |
| **实验时间** | **2020.10.22** | | **实验地点** | **A主410** | |
| **实验成绩** |  | | **实验性质** | **□验证性 □设计性 ■综合性** | |
| 教师评价：  □算法/实验过程正确；□源程序/实验内容提交 □程序结构/实验步骤合理；  □实验结果正确； □语法、语义正确； □报告规范；  其他：  评价教师签名： | | | | | |
| 一、实验目的  🡪了解编译器中语法分析环节的作用和意义  🡪掌握语法分析的方法和步骤  🡪熟悉自底向上、自顶向下等各类语法分析方法；掌握语法树的构造方法 | | | | | |
| 二、实验项目内容  一）实验内容  设计并实现实验一中的C语言子集的语法分析程序，要求实现以下功能：   1. 必做项： 2. 可以识别出用C语言子集中数据类型和语句集编写的源程序，并以语法分析树的形式输出分析结果。   （基本数据类型至少包含整数、布尔类型；至少包含赋值语句、变量申明语句、算数加/减法表达式、布尔表达式、循环语句、分支语句）   1. 检查源程序中存在的语法错误，并报告错误所在的位置。 2. 可选项：   需在给定C语言子集的基础上进行扩充,实现下列要求：   1. 能识别函数调用语句 2. 对源程序中出现的错误进行适当的恢复，使得语法分析可以继续进行，检查并报告源程序中存在的所有语法错误及错误位置   二）实验要求：   1. 自由选用程序设计语言（java，python，c/c++）作为实现语言，手工编写语法分析程序。 2. 提交实验报告及源代码。实验报告需严格遵循学校文档规范，内容包含对应文法、语法分析测试用例。   三）C语言子集  数据类型: **int**, 无符号整数, 取值范围0-9999  int a;  int a,b;  int a = 1;  算术运算符: **+**,**-**  a = b + 1;  a = b + c;  赋值运算符: **=**  a = 1;  关系运算符: **==** ,**>**,**<,<>,>=,<=**  a = (b==c);  a = (b>c);  a = (b<c);  逻辑运算符: **&&**, **||, !**  a = (b&&c);  a = (b||c);  a = (!b);  条件语句: **if**  if(a==b)  {  };  循环语句: **while**  while(a==b)  {  };  输入,输出: **get**,**put**  get(a);  put(a);  语句结束符: **;**  **加分项：**  条件语句 **if else**  if(a==b)  {  };  else  {  }; | | | | | |
| 三、实验过程或算法 | | | | | |
| 1. 语言说明   设计的语法分析器包括了上面列出的C语言子集的所有项目，我自己设计了一个没有左递归，提取了左公因式的LL(1)文法，文法具体如下所示。    其中，终结符对应的编号和词法分析一致，这里给出所有非终结符的编号表。   |  |  | | --- | --- | | 非终结符 | 编号 | | 程序 | 100 | | 句子 | 101 | | 定义左公因式 | 102 | | 定义可赋值 | 103 | | 赋值语句 | 104 | | 加变量 | 105 | | 表达式 | 106 | | 可选二元运算部分 | 107 | | 一元运算符 | 109 | | 二元运算符 | 110 | | 数 | 111 | | 输入语句 | 112 | | 输出语句 | 113 | | 循环语句 | 114 | | 条件语句 | 115 | | else部分 | 116 |  1. 选用语法分析方法以及相应设计   本次实验我选择了LL(1)分析方法，使用C++进行编写。  针对上面的文法，我求出了所有非终结符的Nullable集合，First集合，Follow集合以及针对产生式的First\_S集合，并构建了分析表。  这里说明一下，我这里写的LL(1)分析方法和书上有所不同，我并没有在First和Follow集里添加ε元素，而是求了一个Nullable集合来求出所有可以为空的非终结符。First集合和Follow集合和书上定义相同。First\_S集合指的是对于每一个产生式的右部求得的First集合。  其中，各个集合的求法以及分析表的求法以及结果如下所示。   1. Nullable集合   //获得所有的NULLABLE的非终结符  unordered\_set<int> getNullable(){      unordered\_set<int> nullable;      while(1){          int len1 = nullable.size();          for(auto production:productions){              int left = production.left;              vector<int> right = production.right;              if(right.size() == 0){                  nullable.insert(left);              }              int allNullable = 1;              for(auto icon:right){                  if(nullable.count(icon) == 0)                  {                      allNullable = 0;                      break;                  }              }              if(allNullable){                  nullable.insert(left);              }          }          int len2 = nullable.size();          if(len2 == len1){              break;          }      }      return nullable;  }  Nullable求得的结果如下所示：     1. First集合   //获得非终结符的First集  unordered\_map<int,unordered\_set<int> > getFirst(unordered\_set<int> nullable){      //计算first集需要传入nullable      unordered\_map<int,unordered\_set<int>> first;      while(1){          //先计算first集的大小          int size1 = 0;          for(auto x:first){              size1 += x.second.size();          }          //更新first集          for(auto production:productions){              int left = production.left;              vector<int> right = production.right;              if(right.size() > 0){                  for(int icon:right){                      if(isTerminal(icon)){                          first[left].insert(icon);                          break;                      }                      if(isNonTerminal(icon)){                          //first[left] = first[left] 并 first[icon]                          for(auto x:first[icon]){                              first[left].insert(x);                          }                          if(nullable.count(icon) == 0){ //icon不是nullable，直接结束                              break;                          }                      }                  }              }          }          //再次计算first集大小          int size2 = 0;          for(auto x:first){              size2 += x.second.size();          }          if(size2 == size1){              return first;          }      }  }  First集合求得的结果如下所示：     1. Follow集合   //获得非终结符的Follow集  unordered\_map<int,unordered\_set<int> > getFollow(unordered\_set<int> nullable,unordered\_map<int,unordered\_set<int>> first){      unordered\_map<int,unordered\_set<int> > follow;      follow[100].insert(200);  //起始符的follow集初始化为终止符      while(1){          //先计算follow集的大小          int size1 = 0;          for(auto x:follow){              size1 += x.second.size();          }          //更新follow集          for(auto production:productions){              int left = production.left;              vector<int> right = production.right;              unordered\_set<int> temp = follow[left];              int rightLen = right.size();              for(int i=rightLen-1;i>=0;i--){                  int icon = right[i];                  if(isTerminal(icon)){                      temp.clear();                      temp.insert(icon);                  }                  else{                      //follow[icon] 并= temp  哇，这里之前写错了，找了半天的错                      for(auto x:temp){                          follow[icon].insert(x);                      }                      if(nullable.count(icon) == 0) //icon不是nullable                      {                          temp = first[icon];                      }else{                          //temp 并= first[icon]                          for(auto x:first[icon]){                              temp.insert(x);                          }                      }                  }              }          }          //再计算follow集大小          int size2 = 0;          for(auto x:follow){              size2 += x.second.size();          }          if(size1 == size2) return follow;      }  }  Follow集合求得的结果如下所示：     1. First\_S集合   //获得first\_s集合(针对每一个产生式的first集)  unordered\_map<int,unordered\_set<int>> getFirstS(unordered\_set<int> nullable,unordered\_map<int,unordered\_set<int>> first,unordered\_map<int,unordered\_set<int>> follow){      unordered\_map<int,unordered\_set<int>> firstS;      int index = 0;      for(auto production:productions){          int left = production.left;          vector<int> right = production.right;          int middleReturn = 0;          for(int icon:right){              //icon是终结符              if(isTerminal(icon)){                  firstS[index].insert(icon);                  middleReturn = 1;                  break;              }              //icon是非终结符              else{                  //firstS[index] 并= first[icon]                  for(int x:first[icon]){                      firstS[index].insert(x);                  }                  //如果icon不是nullable                  if(nullable.count(icon)==0){                      middleReturn = 1;                      break;                  }              }          }          if(!middleReturn){              //如果没有中途返回，firstS[index] 并= follow[left]              for(int x:follow[left]){                  firstS[index].insert(x);              }          }          index++;      }      return firstS;  }    其中，左边是产生式的index,右边是产生式的First集合。   1. 分析表   //构建分析表  void createAnalyzeTable(unordered\_map<int,unordered\_set<int>> firstS){      for(int i=0;i<200;i++)          for(int j=0;j<200;j++)              analyze\_table[i][j].clear();        for(auto x:firstS){          int productionIndex = x.first;          int NT = productions[productionIndex].left;          for(auto a:x.second){              int T = a;              analyze\_table[NT][T].insert(productionIndex);          }      }  }  分析表构建的结果过大，这里就不列举了，不过，需要对分析表的结果进行合法性的判断，代码如下所示。  bool checkAnalyzeTable(){      //检查table里面有没有冲突的情况      for(int nt:nonTerminal){          for(int t:terminal){              if(analyze\_table[nt][t].size() > 1)              {                  cout<<"当非终结符"<<" "<<id\_string\_table[nt];                  cout<<" 遇上"<<"终结符 "<<id\_string\_table[t]<<" 时";                  cout<<"，产生式有以下几个选择："<<endl;                  for(int productionIndex:analyze\_table[nt][t]){                      printProduction(productionIndex);                  }                  return false;              }          }      }      cout<<"分析表没有冲突问题"<<endl;      return true;  }   1. 数据结构说明   由上述的代码可以看出，我使用了unoredered\_set来存放Nullable集合，使用了unordered\_map来存放First,Follow,First\_S集合。  本实验中我构建了一个production的结构体来存放文法，TreeNode结构体来存放语法树，grammerAnalyzeResult结构体来存放语法分析的返回结果。  struct production{      int left; //左部是一个编号，对应一个非终结符      vector<int> right; //右部是一个编号序列      production(int left,vector<int> right){          this->left = left;          this->right = right;      }  };  struct TreeNode{      int number; // 对应编号      vector<TreeNode\*> sons;  };  struct grammerAnalyzeResult{      //语法分析返回的结果一个是语法树，一个是错误的位置      TreeNode\* grammerTree;      int tokenWrongIndex;  };  使用vector来存放文法，unordered\_set来存放终结符和非终结符，map来存放符号表以及符号翻译表，unordered\_set来存放分析表。  vector<production> productions; //文法  unordered\_set<int> nonTerminal;  //非终结符的集合  unordered\_set<int> terminal;    //终结符的集合  map<string,int> table;   //符号表  <符号,编号>  map<int,string> id\_string\_table; //符号翻译表  unordered\_set<int> analyze\_table[200][200];   1. 语法分析过程代码   LL(1)分析方法使用分析表以及一个分析栈来进行语法分析，其具体代码如下所示,这里只摘取了部分，具体见源代码文件。  //根据分析表进行语法分析      int tokenPos = 0; int tokenLen = tokens.size();      stack<TreeNode\*> s;      TreeNode\* endMark = new TreeNode; endMark->number = 200; //endMark->sons.clear();      TreeNode\* root = new TreeNode; root->number=100; //root->sons.clear();      s.push(endMark);//push结束符      s.push(root); //push开始符号      int error = 0;      cout<<"语法分析过程如下所示："<<endl;      cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;      while(!s.empty()){          //打印当前栈内元素          stack<TreeNode\*> temp = s;          cout<<"栈内元素: ";          while(!temp.empty()){              cout<<id\_string\_table[temp.top()->number]<<" ";              temp.pop();          }          cout<<"栈底";          cout<<"    ";          //打印当前正在处理的token          cout<<"输入: ";          for(int i=tokenPos;i<tokenLen;i++){              cout<<id\_string\_table[tokens[i]];          }          TreeNode\* curNode = s.top();          int x  = s.top()->number;          if(isTerminal(x))          {              if(x == tokens[tokenPos]){              tokenPos++;              s.pop();              }else{                  error = 1;                  cout<<endl;                  result.tokenWrongIndex = tokenPos;                  cout<<"!!!!发生了语法错误!!!!"<<endl;                  break;              }              cout<<endl;          }          else{              s.pop();              unordered\_set<int> productionIndexs = analyze\_table[x][tokens[tokenPos]]; //查表              //虽然是Indexs，但是讲道理，只有一个才是正常的,我这里就当一个算了              if(productionIndexs.size() != 1){                  error = 1;                  cout<<endl;                  if(productionIndexs.size() ==0 )cout<<"!!!!发生了语法错误,分析表对应为空!!!!"<<endl;                  if(productionIndexs.size() >1 )cout<<"!!!!发生了语法错误,分析表对应选择大于1!!!!"<<endl;                  result.tokenWrongIndex = tokenPos;                  break;              }              vector<int> right;              int index = -1;              for(int productionIndex:productionIndexs){                  index = productionIndex;                  right = productions[productionIndex].right;              }              cout<<"   应用文法：";              printProduction(index);                for(int i=right.size()-1;i>=0;i--)              {                  TreeNode\* pushNode = new TreeNode; pushNode->number =right[i]; pushNode->sons.clear();                  s.push(pushNode);                  curNode->sons.push\_back(pushNode); //这里栈要求倒着push，没办法，只好也倒着存进sons了              }            }      }      if(!error) cout<<"语法分析完成无误"<<endl;   1. 语法树以及输出说明   由上面所述，语法树对应TreeNode结构体，一个成员是节点对应的编号，一个成员是所有子孙节点对应的编号向量。  输出对应结构体如下，一个成员是语法树，另一个成员是发生语法错误的位置在token串里的index。  struct TreeNode{      int number; // 对应编号      vector<TreeNode\*> sons;  };    struct grammerAnalyzeResult{      //语法分析返回的结果一个是语法树，一个是错误的位置      TreeNode\* grammerTree;      int tokenWrongIndex;  };   1. 错误处理   本语法分析器可以发现错误并输出错误所在的行号和列号。因为词法分析器输出的结果中包含每一个token的行号和列号，所以只需要得到错误的token在原tokens中的index就可以输出错误的具体位置，如下所示： | | | | | |
| 四、实验测试  1. 样例1(if else展示以及语法树的绘制)  if(a>1){ a = a+1;};  else { a = a - 1;};  语法分析过程以及输出如下所示：  语法分析过程如下所示：  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  栈内元素: 程序 $ 栈底 输入: if(变量>常量){变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：程序 -> 句子 程序  栈内元素: 句子 程序 $ 栈底 输入: if(变量>常量){变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：句子 -> 条件语句  栈内元素: 条件语句 程序 $ 栈底 输入: if(变量>常量){变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：条件语句 -> if ( 表达式 ) { 程序 } ; else部分  栈内元素: if ( 表达式 ) { 程序 } ; else部分 程序 $ 栈底 输入: if(变量>常量){变量=变量+常量;};else{变量=变量-常量;};$  栈内元素: ( 表达式 ) { 程序 } ; else部分 程序 $ 栈底 输入: (变量>常量){变量=变量+常量;};else{变量=变量-常量;};$  栈内元素: 表达式 ) { 程序 } ; else部分 程序 $ 栈底 输入: 变量>常量){变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：表达式 -> 数 可选二元运算部分  栈内元素: 数 可选二元运算部分 ) { 程序 } ; else部分 程序 $ 栈底 输入: 变量>常量){变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：数 -> 变量  栈内元素: 变量 可选二元运算部分 ) { 程序 } ; else部分 程序 $ 栈底 输入: 变量>常量){变量=变量+常量;};else{变量=变量-常量;};$  栈内元素: 可选二元运算部分 ) { 程序 } ; else部分 程序 $ 栈底 输入: >常量){变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：可选二元运算部分 -> 二 元运算符 数  栈内元素: 二元运算符 数 ) { 程序 } ; else部分 程序 $ 栈底 输入: >常量){变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：二元运算符 -> >  栈内元素: > 数 ) { 程序 } ; else部分 程序 $ 栈底 输入: >常量){变量=变量+常量;};else{变量=变量-常量;};$  栈内元素: 数 ) { 程序 } ; else部分 程序 $ 栈底 输入: 常量){变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：数 -> 常量  栈内元素: 常量 ) { 程序 } ; else部分 程序 $ 栈底 输入: 常量){变量=变量+常量;};else{变量=变量-常量;};$  栈内元素: ) { 程序 } ; else部分 程序 $ 栈底 输入: ){变量=变量+常量;};else{变量=变量-常量;};$  栈内元素: { 程序 } ; else部分 程序 $ 栈底 输入: {变量=变量+常量;};else{变量=变量-常量;};$  栈内元素: 程序 } ; else部分 程序 $ 栈底 输入: 变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：程序 -> 句子 程序  栈内元素: 句子 程序 } ; else部分 程序 $ 栈底 输入: 变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：句子 -> 赋值语句  栈内元素: 赋值语句 程序 } ; else部分 程序 $ 栈底 输入: 变量=变量+常量;};else{变量=变量-常量;};$ 应用文法：赋值语句 -> 变量 = 表达式 ;  栈内元素: 变量 = 表达式 ; 程序 } ; else部分 程序 $ 栈底 输入: 变量=变量+常量;};else{变量=变量-常量;};$  栈内元素: = 表达式 ; 程序 } ; else部分 程序 $ 栈底 输入: =变量+常量;};else{变量=变量-常量;};$  栈内元素: 表达式 ; 程序 } ; else部分 程序 $ 栈底 输入: 变量+常量;};else{变量=变量-常量;};$ 应用文法：表达式 -> 数 可选二元运算部分  栈内元素: 数 可选二元运算部分 ; 程序 } ; else部分 程序 $ 栈底 输入: 变量+常量;};else{变量=变量-常量;};$ 应用文法：数 -> 变量  栈内元素: 变量 可选二元运算部分 ; 程序 } ; else部分 程序 $ 栈底 输入: 变量+常量;};else{变量=变量-常量;};$  栈内元素: 可选二元运算部分 ; 程序 } ; else部分 程序 $ 栈底 输入: +常量;};else{变量=变量-常量;};$ 应用文法：可选二元运算部分 -> 二元运算符 数  栈内元素: 二元运算符 数 ; 程序 } ; else部分 程序 $ 栈底 输入: +常量;};else{变量=变量-常量;};$ 应用文法：二元运算符 -> +  栈内元素: + 数 ; 程序 } ; else部分 程序 $ 栈底 输入: +常量;};else{变量=变量-常量;};$  栈内元素: 数 ; 程序 } ; else部分 程序 $ 栈底 输入: 常量;};else{变量=变量-常量;};$ 应用文法：数 -> 常量  栈内元素: 常量 ; 程序 } ; else部分 程序 $ 栈底 输入: 常量;};else{变量=变量-常量;};$  栈内元素: ; 程序 } ; else部分 程序 $ 栈底 输入: ;};else{变量=变量-常量;};$  栈内元素: 程序 } ; else部分 程序 $ 栈底 输入: };else{变量=变量-常量;};$ 应用文法：程序 ->  栈内元素: } ; else部分 程序 $ 栈底 输入: };else{变量=变量-常量;};$  栈内元素: ; else部分 程序 $ 栈底 输入: ;else{变量=变量-常量;};$  栈内元素: else部分 程序 $ 栈底 输入: else{变量=变量-常量;};$ 应用文法：else部分 -> else { 程序 } ;  栈内元素: else { 程序 } ; 程序 $ 栈底 输入: else{变量=变量-常量;};$  栈内元素: { 程序 } ; 程序 $ 栈底 输入: {变量=变量-常量;};$  栈内元素: 程序 } ; 程序 $ 栈底 输入: 变量=变量-常量;};$ 应用文法：程序 -> 句子 程序  栈内元素: 句子 程序 } ; 程序 $ 栈底 输入: 变量=变量-常量;};$ 应用文法：句子 -> 赋值语句  栈内元素: 赋值语句 程序 } ; 程序 $ 栈底 输入: 变量=变量-常量;};$ 应用文法：赋值语句 -> 变量 = 表达式 ;  栈内元素: 变量 = 表达式 ; 程序 } ; 程序 $ 栈底 输入: 变量=变量-常量;};$  栈内元素: = 表达式 ; 程序 } ; 程序 $ 栈底 输入: =变量-常量;};$  栈内元素: 表达式 ; 程序 } ; 程序 $ 栈底 输入: 变量-常量;};$ 应用文法：表达式 -> 数 可选二元运算部分  栈内元素: 数 可选二元运算部分 ; 程序 } ; 程序 $ 栈底 输入: 变量-常量;};$ 应用文法：数 -> 变量  栈内元素: 变量 可选二元运算部分 ; 程序 } ; 程序 $ 栈底 输入: 变量-常量;};$  栈内元素: 可选二元运算部分 ; 程序 } ; 程序 $ 栈底 输入: -常量;};$ 应用文法：可选二元运算部分 -> 二元运算符 数  栈内元素: 二元运算符 数 ; 程序 } ; 程序 $ 栈底 输入: -常量;};$ 应用文法：二元运算符 -> -  栈内元素: - 数 ; 程序 } ; 程序 $ 栈底 输入: -常量;};$  栈内元素: 数 ; 程序 } ; 程序 $ 栈底 输入: 常量;};$ 应用文法：数 -> 常量  栈内元素: 常量 ; 程序 } ; 程序 $ 栈底 输入: 常量;};$  栈内元素: ; 程序 } ; 程序 $ 栈底 输入: ;};$  栈内元素: 程序 } ; 程序 $ 栈底 输入: };$ 应用文法：程序 ->  栈内元素: } ; 程序 $ 栈底 输入: };$  栈内元素: ; 程序 $ 栈底 输入: ;$  栈内元素: 程序 $ 栈底 输入: $ 应用文法：程序 ->  栈内元素: $ 栈底 输入: $  语法分析完成无误  其中，将语法树进行层序遍历，输出如下所示：  下面是语法树输出结果:  处于第1层的 程序 -> 句子 程序  处于第2层的 句子 -> 条件语句  处于第2层的 程序 ->  处于第3层的 条件语句 -> if ( 表达式 ) { 程序 } ; else部分  处于第4层的 if是终结符  处于第4层的 (是终结符  处于第4层的 表达式 -> 数 可选二元运算部分  处于第4层的 )是终结符  处于第4层的 {是终结符  处于第4层的 程序 -> 句子 程序  处于第4层的 }是终结符  处于第4层的 ;是终结符  处于第4层的 else部分 -> else { 程序 } ;  处于第5层的 数 -> 变量  处于第5层的 可选二元运算部分 -> 二元运算符 数  处于第5层的 句子 -> 赋值语句  处于第5层的 程序 ->  处于第5层的 else是终结符  处于第5层的 {是终结符  处于第5层的 程序 -> 句子 程序  处于第5层的 }是终结符  处于第5层的 ;是终结符  处于第6层的 变量是终结符  处于第6层的 二元运算符 -> >  处于第6层的 数 -> 常量  处于第6层的 赋值语句 -> 变量 = 表达式 ;  处于第6层的 句子 -> 赋值语句  处于第6层的 程序 ->  处于第7层的 >是终结符  处于第7层的 常量是终结符  处于第7层的 变量是终结符  处于第7层的 =是终结符  处于第7层的 表达式 -> 数 可选二元运算部分  处于第7层的 ;是终结符  处于第7层的 赋值语句 -> 变量 = 表达式 ;  处于第8层的 数 -> 变量  处于第8层的 可选二元运算部分 -> 二元运算符 数  处于第8层的 变量是终结符  处于第8层的 =是终结符  处于第8层的 表达式 -> 数 可选二元运算部分  处于第8层的 ;是终结符  处于第9层的 变量是终结符  处于第9层的 二元运算符 -> +  处于第9层的 数 -> 常量  处于第9层的 数 -> 变量  处于第9层的 可选二元运算部分 -> 二元运算符 数  处于第10层的 +是终结符  处于第10层的 常量是终结符  处于第10层的 变量是终结符  处于第10层的 二元运算符 -> -  处于第10层的 数 -> 常量  处于第11层的 -是终结符  处于第11层的 常量是终结符  将语法树画出，语法树请见实验报告最后一页。  可以看出，语法树很好地展示了语法分析的整个过程。  2. 样例2(简易计算器)  //输入数据num1,num2,op，根据op确定操作进行运算，最后输出运算结果ans  int num1,num2,op,ans;  get(num1,num2,op);  if(op==0)  {  ans = num1 + num2;  };  if(op==1)  {  ans = num1 - num2;  };  if(op==2)  {  ans = num1 & num2;  };  if(op==3)  {  ans = num1 | num2;  };  put(ans);  进行语法分析的结果如下所示(因为结果过长，这里只展示部分截图)：      语法分析的结果是正确无误的。  3. 样例3(跑马灯)  //循环输入op，改变输出结果out，输入0则结束程序  int num0,num1,out,op;  num1 = 3333;  num2 = 6666;  num3 = 9999;  op = 1;  while(op>0)  {  if(op==1)  {  out = num1;  };  if(op==2)  {  out = num2;  };  if(op==2)  {  out = num3;  };  put(out);  get(op);  };  进行语法分析的结果如下所示(因为结果过长，这里只展示部分截图)：      语法分析的结果是正确无误的。  4. 样例4(错误演示)  int a = 1;123  对上述样例进行语法分析结果如下:    可以看出，语法分析器发现了错误并且定位成功。 | | | | | |
| 五、实验总结  LL(1)语法分析器的重点在于First集、Follow集各种集合以及分析表的构建，构建完毕后使用分析栈的方式对token串进行分析即可。  其中，语法分析相关代码在exp2文件夹里，编写的语法分析器存放在LL1GrammerAnalyzer.cpp, 对于语法分析器的测试文件存放在LL1GrammerAnalyzerTest.cpp。整合了词法分析器以及语法分析器并进行分析的文件存放在grammerAnalyze.cpp。  下面整理了一些在编程过程中遇到的一些小问题：  1. 错误定位  因为语法分析器的输入串只是token序列，发现错误时只能发现是哪一个token发生了错误，而不能确定token在代码中的具体位置。所以我给词法分析器的结果中添加了每一个token的位置信息，从而来实现错误定位。  2. 语法树的输出  把语法树打印出来是一件比较麻烦的事情，最后我采用了层序遍历的方式输出了语法树，虽然没有树形结构看起来那么一目了然，但其实也比较清楚明了。 | | | | | |

