|  |
| --- |
| 编译原理实验课  实验报告  **指导教师：**  **年 级： 2020**  **班 级： 32**  **小组编号：**  **组长学号姓名：15200220 李晗**  **组员学号姓名：21200234 木甫提达**  **组员学号姓名：21200223 苑光璞**  **2023年 4 月 10 日**  **计算机科学与技术学院** |

|  |  |  |
| --- | --- | --- |
| **完成实验内容** | | |
| 1）设计并实现 SNL 程序设计语言的编译程序  2）五个模块：  ⚫ 词法分析模块  ⚫ 语法分析模块（递归下降方法）  ⚫ 语法分析模块（LL(1)方法）  ⚫ 语义分析模块  ⚫ 中间代码生成  3）特色  ⚫ 使用QT对编译结果形成界面  ⚫ 语法分析LL（1）方法的Predict集纠错  ⚫ 有中间代码生成模块 | | |
| **小组成员任务完成情况** | | |
| 姓名 | 具体完成任务 | 工作量  百分比 |
| 李晗 | LL1，中间代码生成，写实验报告 | 35% |
| 木甫提达 | 词法分析，前端设计及实现，写实验报告 | 30% |
| 苑光璞 | 递归下降语法分析及语义分析，写实验报告 | 35% |
| **小组成员协作情况** | | |
| 木甫提达完成词法分析，讲结果存在我们整个组商量好的统一的全局变量里。苑光璞和李晗从词法分析结果得到的token序列，分别做递归下降和LL1。语义分析我们事前设定好符号栈的结构，然后同时做语义分析和中间代码生成，并且木甫提达做前端的QT部分，将我们的代码移植到QT中。然后再是李晗对整个程序进行测试。 | | |
| **实验平台与编程语言** | | |
| **实验平台：VS，QT**  **编程语言：c++** | | |
| **实验方案设计** | | |
| **词法分析：**  状态转换图的实现，采用双switch-case方式进行状态的转换，词法分析结果保存到token序列里面。    enum **NonTerminal**  {  Program, ProgramHead, ProgramName, DeclarePart,  TypeDec, TypeDeclaration, TypeDecList, TypeDecMore,  TypeId, TypeName, BaseType, StructureType,  ArrayType, Low, Top, RecType,  FieldDecList, FieldDecMore, IdList, IdMore,  VarDec, VarDeclaration, VarDecList, VarDecMore,  VarIdList, VarIdMore, ProcDec, ProcDeclaration,  ProcDecMore, ProcName, ParamList, ParamDecList,  ParamMore, Param, FormList, FidMore,  ProcDecPart, ProcBody, ProgramBody, StmList,  StmMore, Stm, AssCall, AssignmentRest,  ConditionalStm, StmL, LoopStm, InputStm,  InVar, OutputStm, ReturnStm, CallStmRest,  ActParamList, ActParamMore, RelExp, OtherRelE,  Exp, OtherTerm, Term, OtherFactor,  Factor, Variable, VariMore, FieldVar,  FieldVarMore, CmpOp, AddOp, MultOp  };  enum **Terminal**  {  ENDFILE, ERROR,  PROGRAM, PROCEDURE, TYPE, VAR, IF,  THEN, ELSE, FI, WHILE, DO, ENDWH,  BEGIN, END1, READ, WRITE, ARRAY, OF,  RECORD, RETURN1,  INTEGER, CHAR, //INTEGER 是integer  ID, INTC, CHARC,  ASSIGN, EQ, LT, PLUS, MINUS,  TIMES, OVER, LPAREN, RPAREN, DOT,  COLON, SEMI, COMMA, LMIDPAREN, RMIDPAREN,  UNDERANGE  };  typedef Terminal LexType;  struct **Word**  {  string str;  LexType tok;  };  struct **Token** {  int line;  Word wd;  int index = -1;  };  **语法分析：**  采用递归下降，LL(1)语法分析方法。其中语法树主要采用课本上的树的结构，如下所示：  语法树：设G是给定的语法，称满足下列条件的树为G的一棵语法树：  1. 树的每个节点都标有G的一个语法符号，且根节点标有初始符S。  2. 如果一个非叶节点A按从左到右顺序有n个儿子节点B1、B2、…、Bn，则: A->B1B2…Bn 一定是G的一个产生式。    extern AnalysisStack\* anlsstack;  extern TreeNode\* treeroot;  extern TreeNode\* root;  extern int LL1Tbl[104][104]; // LL1  extern TreeNode\* currentTree;  extern Token\* currentToken;  extern void **initLL1**();  extern void **pAddChild**(int istmnl, int lex, string name, int childindex);  extern void **doSthInLL1**(int num);  extern void **printErrorMsgLL1**(string reason);  extern string outstrLL1[512]; //  extern int strlineLL1;  extern void **printTreeLL1**(TreeNode\* t, int layer, bool islastson);  extern void **processAddChild**(int istmnl, int lex, string name, int childindex);  对于课设PPT上的部分的节点设置如下所示：    对于课本/PPT上的语法树（本页第一张图）则不如要怎么多信息。多出来的信息便于语义分析或中间代码生成。  **递归下降：**  从调用总程序的分析子程序开始，根据产生式及其Predict集递归调用其他分析子函数，直至分析完成。    **LL1：**  总体设计如下所以：    课设PPT上的要求的语法树的部分在LL1中的操作如下所示。而对于课本/PPT上的语法树，仅仅需要在匹配上的时候将其挂到树上即可。    **值得一提**的是，课设的书上对应的predict集合有误，部分纠正后的结果如下所示：    **语义分析**    我们的符号表采用**驻留法**，检查流程如上图所示，核心代码如下所示。  struct SymbolRecord  {  string name; // 名字  string kind; // 三种类型  string type; // 具体类型  int dir; //访问方式  Token\* tk; // 指向tokenlist  SymbolTable\* next;  };  class SymbolTable  {  public:  SymbolTable()  {  index = 0;  paramcount = 0;  }  SymbolRecord\* table[5120];  int index;  int paramcount;  SymbolTable\* pre;  void addRecord(string name, string kind, string type, Token\* tk, SymbolTable\* next = NULL, int dir = 1)  {  table[index] = new SymbolRecord();  table[index]->name = name;  table[index]->kind = kind;  table[index]->type = type;  table[index]->tk = tk;  table[index]->dir = dir;  table[index]->next = next;  if (next) {  next->pre = this;  }  index++;  }  void printTable(int layer = 0)  {  if(layer==0) cout << "============================= SymbolTable =============================" << endl;  string outputstr;  for (int i = 0; i < index; i++)  {  for (int i = 0; i < layer; i++)  {  cout << "\t";  outputstr += "\t";  }  cout << table[i]->name  << "\t"  << table[i]->kind  << "\t"  << table[i]->type << endl;  outputstr += table[i]->name;  outputstr += "\t";  outputstr += table[i]->kind;  outputstr += "\t";  outputstr += table[i]->type;  outputstr += "\n";  if (table[i]->next)  {  table[i]->next->printTable(layer + 1);  }  }  }  };  struct SymbolBaseNode{  string name; // 名字  string btype; // 具体类型  SymbolBaseNode(string n, string bt) {  name = n;  btype = bt;  }  };  struct SymbolRecNode {  string name; // 名字  string type; // 类型  string btype;  SymbolRecNode(string n, string t, string bt = "") {  name = n;  type = t;  btype = bt;  }  };  struct SymbolRec { //record类型  string RName;  vector<SymbolRecNode\*> SRec;  SymbolRec(string rn) {  RName = rn;  }  };  首先对最外层函数进行处理，识别类型定义和变量声明，别检查相关错误，确认无误后添加到符号表，然后逐个检查内层函数，并构建对应符号表；然后依次检查函数调用（包括参数个数、类型不匹配，函数未定义）及“read”“write”（判断是否为变量名）相关错误，赋值号左边的变量类型，赋值符号左右的类型，以及array类型（包括定义时错误及越界）相关错误。  为了测试语义分析的准确性，我们设计了19个测试点来进行测试，最后都完美的通过了测试。    **中间代码生成**  中间代码生成与语义分析类似，都是对语法树进行扫描，都是主要分为两个部分，对声明的扫描和对程序体的扫描。中间代码格式为四元式形式。  主要是对程序体部分的操作，先扫描声明部分，如果声明中有过函声明，就需要先进入过函声明中进行扫描。全局存放一个CodeFIle头、尾指针，一个临时变量的偏移、一个标号。在扫描程序体时候判断当前是什么节点 | | |
| **程序界面及运行截图** | | |
|  | | |
| **源程序核心代码** | | |
| // 词法分析  void **Step1**(FILE\* fp) {  freopen("step1.txt", "w", stdout);  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* lexial \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;  fp = fopen("cifa.txt", "r");  lexicalAnalyse(fp);  printTokenList();  fclose(fp);  }  // 递归下降  void **Recursion**() {  freopen("rec.txt", "w", stdout);  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* recursion \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;  TreeNode\* t = program();  if (tokenList[Index].wd.tok == ENDFILE) {  cout << "DGXJ success!" << endl;  printTree(t, 0, true);  }  else cout << "DGXJ defeat!" << endl;  }  // LL1old  void **LL1old**() {  freopen("LL1old.txt", "w", stdout);  // freopen("con", "w", stdout);  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* LL1(old) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;  treeroot = programLL1();  printTreeLL1(treeroot, 0, false);  }  // LL1new  void **LL1new**(Node\* & miao) {  freopen("LL1new.txt", "w", stdout);  // freopen("con", "w", stdout);  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* LL1(new) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;  Index = 0;  initLL1();  ParserLL1 test;  SyntaxTree\* t\_root = test.run();  Node\* res = t\_root->getRoot();  if (is\_newLL1\_correct)  printTreeNewLL1(res, 0, true);  miao = res;  }  // 语义分析  void STEP3()  {  TreeNode\* t = treeroot;  freopen("con", "w", stdout);  // freopen("wocaonima.txt", "w", stdout);  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Semantic \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;  t = t->child[1];  for (int i = 0; i < t->child.size() && t->child[i]; i++)  {  TreeNode\* temp = t->child[i];  if (temp->name == "TypeDec")  typeSaveTable(temp, smbltable);  else if (temp->name == "VarDec")  varSaveTable(temp, smbltable);  else if (temp->name == "ProcDec")  {  SymbolTable\* p = NULL;  procSaveTable(temp, smbltable, p);  // cout << "here"<< endl;  }  }  Check\_STEP3(treeroot, smbltable);  SymbolTable\* head = new SymbolTable();  head->addRecord(treeroot->child[0]->child.back()->child[0]->name, "procKind", "program", t->tk, smbltable);  head->printTable();    if(is\_correct\_semantic == 0)  longjmp(jump\_buffer, 1);  }  // 中间代码生成  void **MID**(Node\* t)  {  freopen("mid.txt", "w", stdout);  if (!t)  return;  processMid(t);  printMidCode(*midtable*);  }  void MainWindow::**on\_pushButton\_clicked**()  {  QString str;  QFile file;  file.setFileName("cifa.txt");  file.*open*(QIODevice::WriteOnly );  QTextDocument\* doc = ui->plainTextEdit->document(); //将plainTextEdit中的内容读取到doc中  int count = doc->blockCount(); //计算plainTextEdit中的文本块数(行数)count  for(int i = 0; i < count; i++){  QTextBlock textLine = doc->findBlockByNumber(i); //依次读取文本块至textLine中  str = textLine.text();//依次获取文本块中的字符串  std::string res2 = str.toStdString()+'\n';//string  const char\* res3 = res2.c\_str();//char\*  file.write(res3);  }  file.*close*();  if (setjmp(jump\_buffer) == 0) // 保存下来当前的环境变量  {  freopen("step1.txt", "w", stdout);  freopen("rec.txt", "w", stdout);  freopen("LL1old.txt", "w", stdout);  freopen("LL1new.txt", "w", stdout);  freopen("step3.txt", "w", stdout);  freopen("mid.txt", "w", stdout);  freopen("con", "w", stdout);  FILE\* fp = nullptr;  Step1(fp);  Recursion();  LL1old();  Node\* miao;  LL1new(*miao*);  STEP3();  MID(miao);  } | | |