** **

**课程实验报告**

课程名： 编译原理课程实践

学 院： 数学与计算机学院 系 计算机科学与技术系

专 业： 计算机科学与技术(卓越工程师)

班 级： 计算机科学与技术(卓越工程师) 221班

学 号： 5418122020

姓 名： 马星

任课教师： 廖频

授课学期： 2023 年~~~~ 2024 年 秋季 学期

**目 录**

[**实验一 文法的读入和输出** 5](#_Toc186537166)

[一、实验项目名称 5](#_Toc186537167)

[二、实验目的 5](#_Toc186537168)

[三、实验基本原理 5](#_Toc186537169)

[四、主要仪器设备及耗材 5](#_Toc186537170)

[五、实验步骤（完整内容见光盘） 6](#_Toc186537171)

[1、根据文法定义，设计出文法数据结构Grammar.h 6](#_Toc186537172)

[2、文法数据结构的实现 Grammar.cpp 7](#_Toc186537173)

[3、编写调试文法读入和输出程序 10](#_Toc186537174)

[4、测试程序运行效果：从文本文件中读入一个文法，在屏幕上输出，检查输出结果。 11](#_Toc186537175)

[六、实验数据及处理结果 11](#_Toc186537176)

[七、思考讨论题或体会或对改进实验的建议 12](#_Toc186537177)

[(1)思考题 12](#_Toc186537178)

[(2)实验总结 13](#_Toc186537179)

[八、参考资料 13](#_Toc186537180)

[**实验二 词法分析程序的设计** 14](#_Toc186537181)

[一、实验项目名称 14](#_Toc186537182)

[二、实验目的 14](#_Toc186537183)

[三、实验基本原理 14](#_Toc186537184)

[四、主要仪器设备及耗材 14](#_Toc186537185)

[五、实验步骤（完整内容见光盘） 15](#_Toc186537186)

[1、根据正规式，画出状态转换图； 15](#_Toc186537187)

[2、根据状态图，设计词法分析算法； 15](#_Toc186537188)

[3、采用C或C++语言，设计函数scan( )，实现该算法； 19](#_Toc186537189)

[4、编制测试程序（主函数main）； 19](#_Toc186537190)

[5、调试程序：读入文本文件，检查输出结果。 20](#_Toc186537191)

[六、实验数据及处理结果 22](#_Toc186537192)

[七、思考讨论题或体会或对改进实验的建议 22](#_Toc186537193)

[(1) 思考题 22](#_Toc186537194)

[(2) 实验体会 23](#_Toc186537195)

[八、参考资料 23](#_Toc186537196)

[**实验三 LL（1）文法构造** 24](#_Toc186537197)

[一、实验项目名称 24](#_Toc186537198)

[二、实验目的 24](#_Toc186537199)

[三、实验基本原理 24](#_Toc186537200)

[四、主要仪器设备及耗材 25](#_Toc186537201)

[五、实验步骤（完整内容见光盘） 25](#_Toc186537202)

[1、学习LL（1）文法的分析条件； 25](#_Toc186537203)

[2、学习构造LL（1）文法的算法； 25](#_Toc186537204)

[3、结合实验1给出的数据结构，编程实现构造LL（1）文法的算法； 26](#_Toc186537205)

[4、结合实验1编程和调试实现对一个具体文法运用上述算法，构造它的LL（1）文法形式； 30](#_Toc186537206)

[5、把实验结果写入一个新建立的文本文件。 30](#_Toc186537207)

[六、实验数据及处理结果 31](#_Toc186537208)

[七、思考讨论题或体会或对改进实验的建议 32](#_Toc186537209)

[(1) 思考题 32](#_Toc186537210)

[(2) 实验体会 32](#_Toc186537211)

[八、参考资料 32](#_Toc186537212)

[**实验四 递归下降分析程序** 33](#_Toc186537213)

[一、实验项目名称 33](#_Toc186537214)

[二、实验目的 33](#_Toc186537215)

[三、实验基本原理 33](#_Toc186537216)

[四、主要仪器设备及耗材 33](#_Toc186537217)

[五、实验步骤（完整内容见光盘） 33](#_Toc186537218)

[1、分析文法，将给出的文法转化为LL(1)文法； 33](#_Toc186537219)

[2、学习递归下降分析程序的结构，设计合理的递归下降分析程序； 34](#_Toc186537220)

[3、编写测试程序，包括表达式的读入和结果的输出； 39](#_Toc186537221)

[4、测试程序运行效果，测试数据可以参考下列给出的数据。 40](#_Toc186537222)

[六、实验数据及处理结果 43](#_Toc186537223)

[七、思考讨论题或体会或对改进实验的建议 43](#_Toc186537224)

[(1) 思考题 43](#_Toc186537225)

[(2) 实验体会 43](#_Toc186537226)

[八、参考资料 44](#_Toc186537227)

[**实验五 逆波兰式的翻译和计算** 45](#_Toc186537228)

[一、实验项目名称 45](#_Toc186537229)

[二、实验目的 45](#_Toc186537230)

[三、实验基本原理 45](#_Toc186537231)

[四、主要仪器设备及耗材 45](#_Toc186537232)

[五、实验步骤（完整内容见光盘） 45](#_Toc186537233)

[(1) 拓广文法 46](#_Toc186537234)

[(2) 构建能识别活前缀的DFA 47](#_Toc186537235)

[(3) 构建Action-Goto表, 使用SLR(1) 48](#_Toc186537236)

[(4) 语义动作读入和预处理 50](#_Toc186537237)

[(5) 输入表达式, 自底向上构建语法树 54](#_Toc186537238)

[(6) 计算属性 57](#_Toc186537239)

[(7) 主函数测试 59](#_Toc186537240)

[(8) 程序运行结果 61](#_Toc186537241)

[六、实验数据及处理结果 63](#_Toc186537242)

[七、思考讨论题或体会或对改进实验的建议 63](#_Toc186537243)

[(1) 思考题 63](#_Toc186537244)

[(2) 实验体会 64](#_Toc186537245)

[八、参考资料 64](#_Toc186537246)

[**实验六 语法制导的三地址代码生成** 65](#_Toc186537247)

[一、实验项目名称 65](#_Toc186537248)

[二、实验目的 65](#_Toc186537249)

[三、实验基本原理 65](#_Toc186537250)

[四、主要仪器设备及耗材 66](#_Toc186537251)

[五、实验步骤（完整内容见光盘） 66](#_Toc186537252)

[1、考虑给定的文法，消除左递归，提取左因子。 66](#_Toc186537253)

[2、编制并化简语法图 66](#_Toc186537254)

[3、编制各个递归子程序函数 69](#_Toc186537255)

[4、连接实验一的词法分析函数scan( )，进行测试 72](#_Toc186537256)

[5、设计三地址代码生成的数据结构和算法 73](#_Toc186537257)

[6、将各个递归子程序函数改写为代码生成函数 74](#_Toc186537258)

[7、调试程序：输入一个语句，检查输出的三地址代码 81](#_Toc186537259)

[六、实验数据及处理结果 82](#_Toc186537260)

[七、思考讨论题或体会或对改进实验的建议 84](#_Toc186537261)

[(1) 思考题 84](#_Toc186537262)

[(2) 实验体会 84](#_Toc186537263)

[八、参考资料 85](#_Toc186537264)

# **实验一 文法的读入和输出**

学生姓名：马星 学 号：5418122020 专业班级： 计算机科学与技术(卓越工程师)221班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期：2024/9/27 实验成绩：

## 一、实验项目名称

熟悉文法的结构，了解文法在计算机内的表示方法。

## 二、实验目的

1. 设计一个表示文法的数据结构；
2. 从文本文件中读入文法，利用定义的数据结构存放文法，并输出；
3. 本实验结果还将用于实验3。

## 三、实验基本原理

1. 了解文法定义的4个部分：

G（Vn, Vt, S, P）

Vn 文法的非终结符号集合，在实验中用大写的英文字母表示；

Vt 文法的终结符号集合，在实验中用小写的英文字母表示；

S 开始符号，在实验中是Vn集合中的一个元素；

P 产生式，分左部和右部，左部为非终结符号中的一个，右部为终结符号或非终结符号组成的字符串，如S->ab|c

1. 根据文法各个部分的性质，设计一个合理的数据结构用来表示文法，
   1. 若使用C语言编写，则文法可以设计成结构体形式，结构体中应包含上述的4部分，
   2. 若使用C++语言编写，则文法可以设计成文法类形式，类中至少含有4个数据成员，分别表示上述4个部分

文法数据结构的具体设计由学生根据自己想法完成，并使用C或C++语言实现设计的数据结构。

1. 利用完成的数据结构完成以下功能：
   1. 从文本文件中读入文法（文法事先应写入文本文件）；
   2. 根据文法产生式的结构，分析出文法的4个部分，分别写入定义好的文法数据结构的相应部分；
   3. 整理文法的结构；
   4. 在计算机屏幕或者文本框中输出文法，文法输出按照一个非终结符号一行，开始符号引出的产生式写在第一行，同一个非终结符号的候选式用“|”分隔的方式输出。

## 四、主要仪器设备及耗材

PC微机

DOS操作系统或 Windows 操作系统

Turbo C 程序集成环境或 Visual C++ 程序集成环境

## 五、实验步骤（完整内容见光盘）

### 1、根据文法定义，设计出文法数据结构Grammar.h

文法由 开始符、非终结符集合、终结符集合、产生式集合 这四部分组成

开始符使用string存储

非终结符集合、终结符集合使用set<string>存储

产生式分为左部和右部集合, 左部使用string存储, 右部使用vector<string>存储, 则右部集合为vector< vector<string>>, 使用哈希映射左部->右部集合, 即产生式集合使用map<string, vector<vector<string>>>存储

/\*

\* 文法的读入和输出

\*/

class Grammar {

public:

explicit Grammar();

string S;//开始符

set<string> Vn;//非终结符

set<string> Vt;//终结符

map<string, vector<vector<string>>> P;// left -> right1|right2|...

string sourceCode;// 原始文本

/\*\*

\* 将文本转为产生式, 存储在类对象内

\* @param input 输入的产生式文本

\*/

void scan(const string &input);

/\*\*

\* 根据是否大写区分非终结符和终结符

\* @param Vn 非终结符集合

\* @param Vt 终结符集合

\* @param s 字符, 如果是非终结符则放入Vn, 否则放入Vt

\*/

static void addToSet(set<string> &Vn, set<string> &Vt, const string &s);

/\*\*

\* 读取文件文本, 并转为产生式

\*/

void readFromFile(const string &filename);

vector<string> getVn() const;

vector<string> getVt() const;

// 打印信息的方法

void printSourceCode() const;

void printVn() const;

void printVt() const;

void printS() const;

void printP() const;

void print() const;

};

### 2、文法数据结构的实现 Grammar.cpp

使用scan函数, 将读入的文本产生式转换为上面定义的文法四个部分

然后提供print方法打印这四个部分

#include "Grammar.h"

Grammar::Grammar() = default;

void Grammar::scan(const string &input) {

istringstream stream(input);

string line;

while (getline(stream, line)) {// 按行分隔

if (!line.empty() && (line.back() == '\n' || line.back() == ';')) line.pop\_back();

// 按->分隔

size\_t arrowPos = line.find("->");

if (arrowPos == string::npos) continue;

string leftPart = line.substr(0, arrowPos), rightPart = line.substr(arrowPos + 2);

if (S.empty())S = leftPart;

addToSet(Vn, Vt, leftPart);// 记录Vn、Vt

// 产生式右侧按|分隔

istringstream rightStream(rightPart);

string option;

while (getline(rightStream, option, '|')) {

vector<string> r; // 拆分option中的每一项到vector中, 方便区分

for (char ch: option) r.emplace\_back(1, ch);

P[leftPart].push\_back(r);

addToSet(Vn, Vt, option);

}

}

}

/\*\*

\* 大写放Vn,其余放Vt

\*/

void Grammar::addToSet(set<string> &Vn, set<string> &Vt, const string &s) {

for (char c: s) {

string x(1, c);

if (isupper(c)) {

Vn.insert(x);

} else {

Vt.insert(x);

}

}

}

void Grammar::readFromFile(const string &filename) {

sourceCode = read(filename);

scan(sourceCode);

}

vector<string> Grammar::getVn() const {

return {Vn.begin(), Vn.end()};

}

vector<string> Grammar::getVt() const {

return {Vt.begin(), Vt.end()};

}

void Grammar::printSourceCode() const {

cout << "文件文本:\n" << sourceCode << endl;

}

void Grammar::printVn() const {

cout << "非终结符:{";

for (const string &vn: Vn) cout << vn << ",";

if (!Vn.empty()) cout << "\b";

cout << "}\n";

}

void Grammar::printVt() const {

cout << "终结符:{";

for (const string &vn: Vt) cout << vn << ",";

if (!Vt.empty()) cout << "\b";

cout << "}\n";

}

void Grammar::printS() const {

cout << "开始符号:" << S << "\n";

}

void Grammar::printP() const {

cout << "产生式:{" << "\n";

cout << "\t";

cout << S << "->";

for (size\_t i = 0; i < P.at(S).size(); ++i) {

if (i > 0)cout << "|";

for (const string &j: P.at(S)[i])cout << j;

}

cout << endl;

for (auto &it: P) {

if (it.first == S) continue;

cout << "\t";

cout << it.first << "->";

for (size\_t i = 0; i < it.second.size(); ++i) {

if (i > 0)cout << "|";

for (const string &j: it.second[i])cout << j;

}

cout << endl;

}

cout << "}\n";

}

void Grammar::print() const {

printSourceCode();

printVn();

printVt();

printS();

printP();

}

### 3、编写调试文法读入和输出程序

输入数据：

编辑一个文本文文件g.txt，在文件中输入如下内容：

S->Qc;

S->c;

Q->Rb;

Q->b;

R->Sa;

R->a;

正确结果：

上述文法整理后的输出形式：

S->Qc|c;

Q->Rb|b;

R->Sa|a;

**测试程序:**

#include <bits/stdc++.h>

using namespace std;

#include "Grammar.h"

int main() {

Grammar grammar;

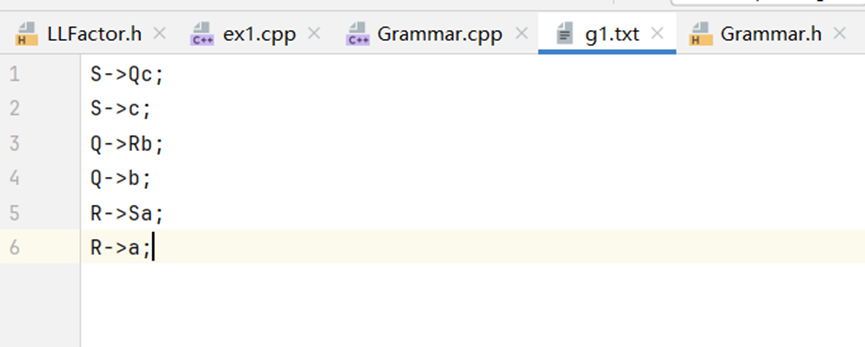
grammar.readFromFile("../Ex1/g1.txt");

grammar.print();

return 0;

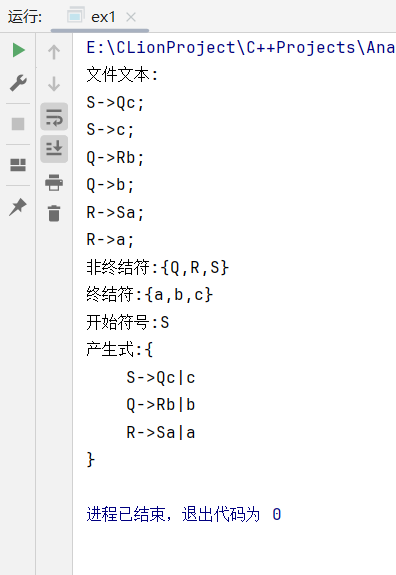
}

**输入文件:**



### 4、测试程序运行效果：从文本文件中读入一个文法，在屏幕上输出，检查输出结果。

**程序输出:**

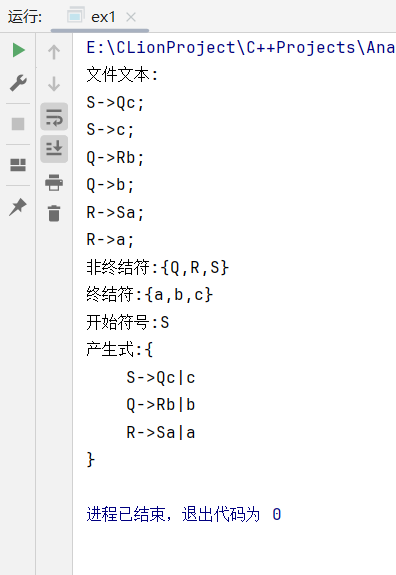


## 六、实验数据及处理结果

成功读取了文件中的产生式文本, 并存储到了定义的文法数据结构中, 对其进行调用也可正常打印结果(如图)

更改输入的文件内容, 结果均正确

当输入的文法不正确时, 程序也会给出相应报错



## 七、思考讨论题或体会或对改进实验的建议

### (1)思考题

**1、 如何让设计的文法结构满足各种文法的要求？**

充分了解并掌握不同类型的文法（如上下文无关文法、正则文法等）的特点和限制。

在设计文法时，尽量使用通用的符号和规则，避免特定于某种文法的专有符号或结构。

对于复杂的文法要求，可以尝试将文法分解为更简单的子文法，然后通过组合这些子文法来满足整体要求。

用文法转换工具或算法，将一种文法形式转换为另一种形式，以适应不同的需求。

**2、 如何设计文法才能跟简单地表示文法，同时又降低程序编写难度？**

可以使用BNF（巴科斯范式）或EBNF（扩展巴科斯范式）作为输入格式, 这些格式已经为文法提供了一种清晰且广泛接受的表示方式，使得解析和转换过程更为直接。

还可以通过提供图形用户界面（GUI）或者命令行工具来辅助用户输入文法，这样可以减少手动编码的错误并提高用户体验。

在程序实现方面，采用模块化设计可以帮助降低复杂性，比如将词法分析器、语法分析器和语义分析器等功能分离到不同的模块中。

### (2)实验总结

通过本次实验，我们学习了如何设计和实现一个用于存储和处理形式语言文法的数据结构。

从文件中读取文法定义，并将其转换为内部表示形式，这有助于我们进一步对文法进行分析和操作。

在实验过程中，遇到了一些挑战，例如如何处理复杂的嵌套结构以及确保数据结构的灵活性以适应不同类型的文法。通过这些问题的解决，我们加深了对形式语言理论的理解，并且提高了编程技能。

此外,还探索了如何优化数据结构以提高效率和可维护性。

总的来说，这次实验不仅掌握了理论知识，还锻炼了实践能力和问题解决能力。

## 八、参考资料

《编译原理课程实践任务书》

**实验二 词法分析程序的设计**

学生姓名：马星 学 号：5418122020 专业班级： 计算机科学与技术(卓越工程师)221班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期：2024/10/11 实验成绩：

## 一、实验项目名称

掌握计算机语言的词法分析程序的开发方法。

## 二、实验目的

编制一个能够分析三种整数、标识符、主要运算符和主要关键字的词法分析程序。

## 三、实验基本原理

1、根据以下的正规式，编制正规文法，画出状态图；

标识符 <字母>(<字母>|<数字字符>)\*

十进制整数 0 | (（1|2|3|4|5|6|7|8|9）（0|1|2|3|4|5|6|7|8|9）\*)

八进制整数 0（1|2|3|4|5|6|7）（0|1|2|3|4|5|6|7）\*

十六进制整数 0x（0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f）（0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f）\*

运算符和界符 + - \* / > < = ( ) ；

关键字 if then else while do

2、根据状态图，设计词法分析函数int scan( )，完成以下功能：

1） 从文本文件中读入测试源代码，根据状态转换图，分析出一个单词，

2） 以二元式形式输出单词<单词种类，单词属性>

其中单词种类用整数表示：

0：标识符

1：十进制整数

2：八进制整数

3：十六进制整数

运算符和界符，关键字采用一字一符，不编码

其中单词属性表示如下：

标识符，整数由于采用一类一符，属性用单词表示

运算符和界符，关键字采用一字一符，属性为空

3、编写测试程序，反复调用函数scan( )，输出单词种别和属性。

## 四、主要仪器设备及耗材

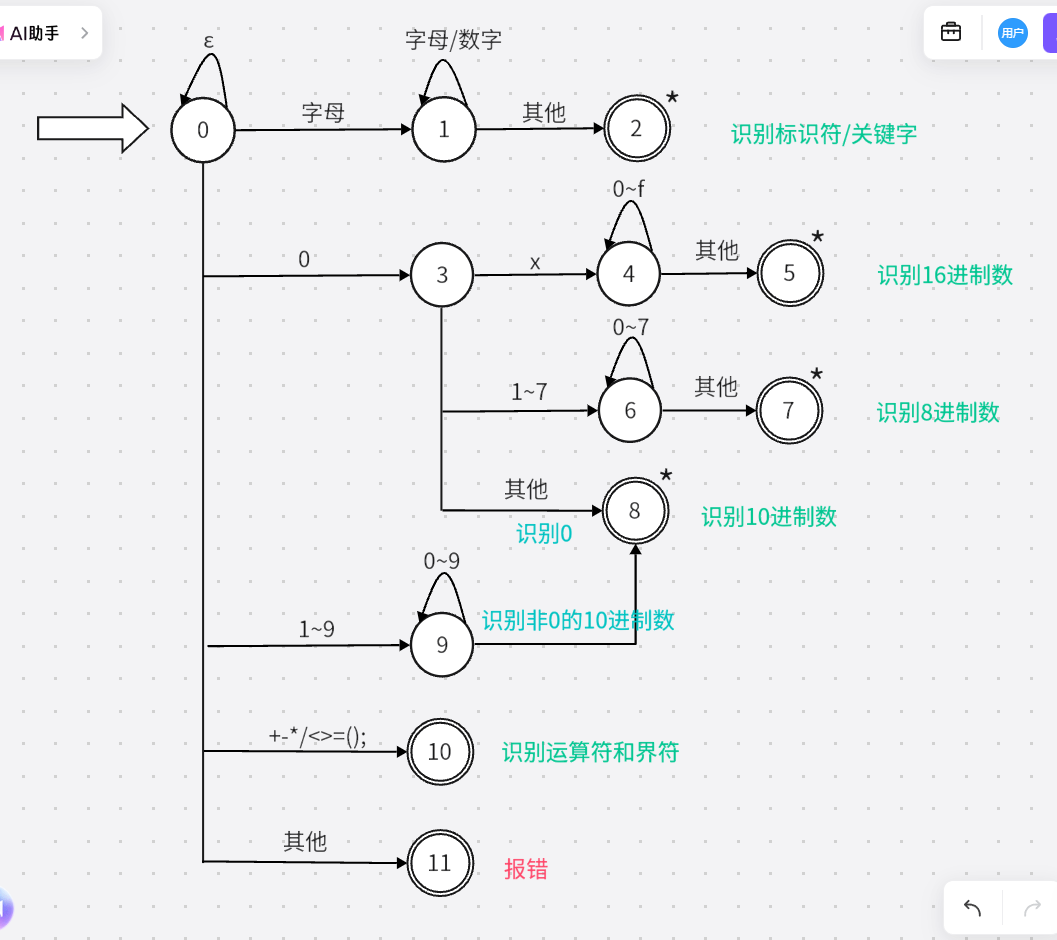
PC微机

DOS操作系统或 Windows 操作系统

Turbo C 程序集成环境或 Visual C++ 程序集成环境

## 五、实验步骤（完整内容见光盘）

### 1、根据正规式，画出状态转换图；



### 2、根据状态图，设计词法分析算法；

#### (1) 定义枚举类表示状态和类型

enum TokenType {

Undefined = -1,//未定义

Identifier = 0, // 标识符

DecInt = 1, // 十进制整数

OctInt = 2, // 八进制整数

HexInt = 3, // 十六进制整数

Symbol = 4, // 运算符和界符

Keyword = 5 // 关键字

};

class Token {

public:

TokenType type;

string value;

}

enum State {

S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11

};

#### (2) 为每个状态定义对应的状态转换程序

对于非终态S0,S1,S3,S4,S6,S9, 定义转换程序

对于终态S2,S5,S7,S8,S10, 读取缓冲区的字符串, 根据到达的终态识别类型, 然后组成Token(类型,值)

对于回退字符态S2,S5,S7, 每次做完一次状态转换, 就判断是否到达这些状态, 如果是, 则将输入指针向前移动一位

对于报错态S11, 打印报错信息到控制台, 并终止程序

##### ①非终态 - 转换程序

static State P\_S0(char ch, string &buffer) {

if (ch == ' ' || ch == '\n' || ch == '\t') {// 接收空白, 仍然为S0

return S0;

} else if (isalpha(ch)) {// 接收字母, 到达S1(标识符或关键字)

buffer += ch;

return S1;

} else if (ch == '0') {// 接收数字0, 到达S3(十进制0或八进制或十六进制)

buffer += ch;

return S3;

} else if ('1' <= ch && ch <= '9') {// 接收非0数字, 到达S9(十进制)

buffer += ch;

return S9;

} else if (isSymbol(ch)) {// 接收算符和界符, 到达S10

buffer += ch;

return S10;

} else {// 不可接收的字符, S11报错

return S11;

}

}

static State P\_S1(char ch, string &buffer) {

if (isalpha(ch) || isdigit(ch)) {// 接收字母或数字,仍为S1

buffer += ch;

return S1;

} else {// 接受其他字符, S2回退

return S2;

}

}

static State P\_S3(char ch, string &buffer) {

if (ch == 'x') {// 接收x, 到达S4(十六进制)

buffer += ch;

return S4;

} else if ('1' <= ch && ch <= '7') {// 接收1~7, 到达S6(八进制)

buffer += ch;

return S6;

} else {// 其他字符, 到达S8, 识别数字0, 并回退

return S8;

}

}

static State P\_S4(char ch, string &buffer) {

if (isdigit(ch) || ('a' <= ch && ch <= 'f')) {// 接收0~f, 仍为S4(十六进制)

buffer += ch;

return S4;

} else {// 其他字符, 到达S5, 识别十六进制数, 并回退

return S5;

}

}

static State P\_S6(char ch, string &buffer) {

if ('0' <= ch && ch <= '7') {// 接收0~7, 仍为S6(八进制)

buffer += ch;

return S6;

} else {// 其他字符, 到达S7, 识别八进制数, 并回退

return S7;

}

}

static State P\_S9(char ch, string &buffer) {

if (isdigit(ch)) {// 接收数字, 仍为S9(十进制)

buffer += ch;

return S9;

} else {// 其他字符, 到达S8, 识别十进制数, 并回退

return S8;

}

}

##### ②终态 - token组装

void putToken(vector<Token> &tokens, State state, const string &buffer) {

Token token;

token.value = buffer;

if (state == S2) {

token.type = isKeyWord(buffer) ? Keyword : Identifier;

} else if (state == S5) {

token.type = HexInt;

} else if (state == S7) {

token.type = OctInt;

} else if (state == S8) {

token.type = DecInt;

} else if (state == S10) {

token.type = Symbol;

} else {

cout << "非法状态, 应为终结态" << endl;

exit(-1);

}

tokens.push\_back(token);

}

##### ③回退字符态和报错态

if (state == S11) {

printf("非法状态S11, 请检查源文件");

exit(-1);

}

if (isBackState(state))i--;// 回退一个字符

if (isEndState(state)) {// 终态

putToken(tokens, state, buffer);//保存buffer中记录的token

state = S0;// 回到初始状态

buffer = "";

}

#### (3) 构建处理程序

class WordFactor {// 词法分析

public:

using P = function<State(char, string &)>;// 处理函数

static map<State, P> m;

static set<char> symbol;// 算符

static set<State> endState;// 终态

static set<State> backState;// 要回退字符的状态

static set<string> keyword;// 关键字

WordFactor() {

symbol = {'+', '-', '\*', '/', '>', '<', '=', '(', ')', ';'};

endState = {S2, S5, S7, S8, S10};

backState = {S2, S5, S7, S8};

keyword = {"if", "then", "else", "while", "do"};

m = {

{S0, P\_S0},

{S1, P\_S1},

{S3, P\_S3},

{S4, P\_S4},

{S6, P\_S6},

{S9, P\_S9}

};

// 输入状态state和下一个字符ch对其进行处理

void deal(char ch, State &state, string &buffer) {

if (isEndState(state)) {

cout << "非法的状态, 应为非终结态" << endl;

exit(-1);

}

state = m[state](ch, buffer);// 执行P\_state, 转换为下一个状态

}

}

### 3、采用C或C++语言，设计函数scan( )，实现该算法；

vector<Token> scan(string sourceCode) {

vector<Token> tokens;// 最终结果

string buffer;// 当前保存的字符串

State state = S0;// 当前状态

int i = 0;// 当前读入字符下标

sourceCode += " ";// 末尾添加一个空白分隔

while (i < sourceCode.size()) {

char ch = sourceCode[i++];// 读入下一个字符

deal(ch, state, buffer);

if (state == S11) {

printf("非法状态S11, 请检查源文件");

exit(-1);

}

if (isBackState(state))i--;// 回退一个字符

if (isEndState(state)) {// 终态

putToken(tokens, state, buffer);//保存buffer中记录的token

state = S0;// 回到初始状态

buffer = "";

}

}

return tokens;

}

### 4、编制测试程序（主函数main）；

#include <bits/stdc++.h>

#include "../CodeHelper/Utils.h"

#include "WordFactor.h"

using namespace std;

int main() {

string sourceCode = read("../Ex2/g2.txt");

cout << "文件内容:" << sourceCode<<endl;

WordFactor wordFactor;

vector<Token> tokens = wordFactor.scan(sourceCode);

cout<<"Tokens:"<<endl;

for (const Token &token: tokens) {

if (token.type == Keyword || token.type == Symbol) {

cout << "<" << token.value << ", " << "-" << ">" << endl;

} else {

cout << "<" << token.type << ", " << token.value << ">" << endl;

}

}

return 0;

}

### 5、调试程序：读入文本文件，检查输出结果。

输入数据：

编辑一个文本文件program.txt，在文件中输入如下内容：

if data+92>0x3f then

data=data+01;

else

data=data-01;

正确结果：

<if , ->

<0 , data>

<+ , ->

<1 , 92>

<> , ->

<3 , 3f>

<then , ->

<0 , data>

<= , >

<0 , data>

<+ , ->

<2 , 1>

<; ,->

<else , ->

<0 , data>

<= , ->

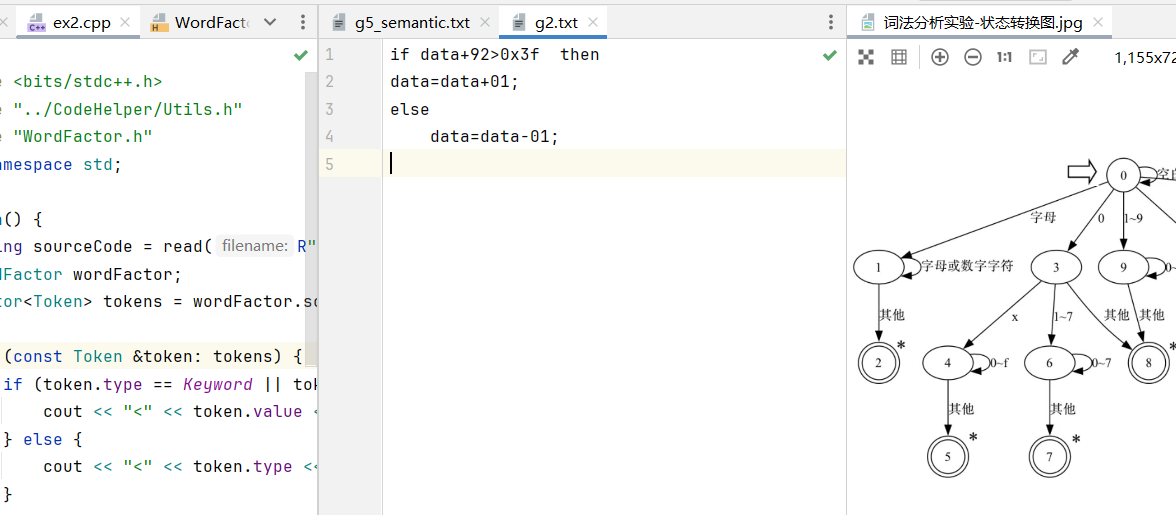
<0 , data>

<- , ->

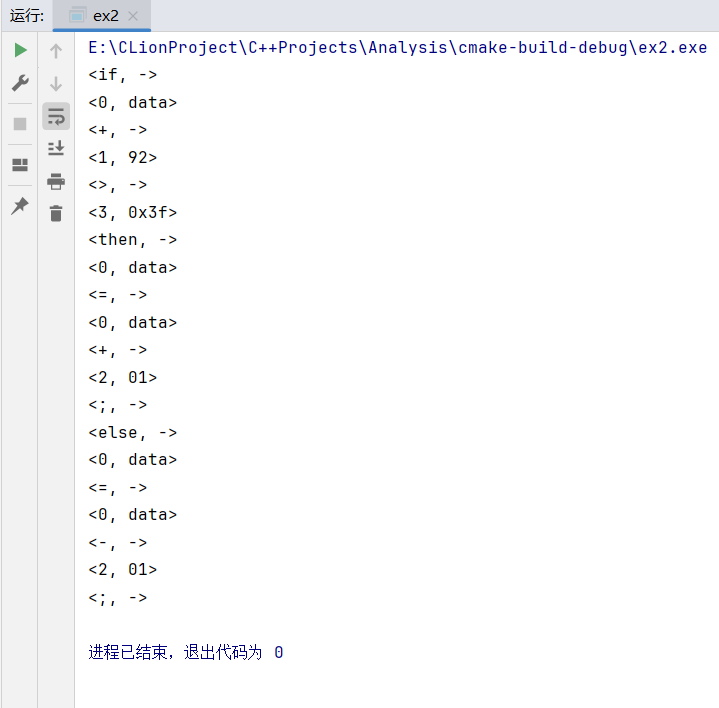
<2 , ->

<; , ->

**输入:**



**输出:**



## 六、实验数据及处理结果

输出与给定答案一致

在输入文件中插入错误符号, 状态到达S11时程序成功进行报错

## 七、思考讨论题或体会或对改进实验的建议

### (1) 思考题

**1、 词法分析能否采用空格来区分单词？**

不能, 仅依赖于空格不足以区分单词

* **多字符运算符和分隔符**: 编程语言中存在多字符的运算符和分隔符，例如C语言中的++（自增）、--（自减）、==（等于）、!=（不等于）等，以及某些语言中的箭头符号->（指向结构体成员）。如果仅以空格作为分隔，这些多字符符号会被错误地分割成单个字符，导致无法正确理解代码意图。
* **字符串字面量与注释**: 在许多编程语言中，字符串内部可以包含空格，但这些空格属于字符串的一部分，不应被视为单词间的分隔。同样，注释内的空格也不应影响单词的划分。
* **变量名中的空格**: 虽然大多数编程语言不允许变量名中包含空格，但仍有特例，如某些脚本语言或配置语言可能支持带有空格的变量名。因此，不能默认所有空格都是单词边界。
* **空白字符处理**: 编译器通常需要忽略多余的空白字符（包括空格、制表符、换行符等），只保留必要的空白以保持源代码的可读性和结构清晰，但这并不意味着空白本身用于定义单词边界。

**2、 程序设计中哪些环节影响词法分析的效率？如何提高效率？**

影响效率的环节:

* **输入读取方式**: 如果每次读取少量字符进行处理，会增加I/O操作次数，降低效率。
* **状态机设计**: 状态机的设计复杂度直接影响匹配速度。复杂的规则可能导致状态跳转频繁，增加处理时间。
* **缓存与重复扫描**: 对于大型文件，多次扫描同一文本区域会浪费时间。
* **数据结构选择**: 存储和查找已识别单词的数据结构（如哈希表、Trie树）的选择也会影响性能。

提高效率的方式:

* **批量读取**: 使用缓冲区一次性读取大块文本，减少I/O操作次数。
* **优化状态机**: 简化状态转换逻辑，尽可能合并相似状态，减少状态跳转次数。对于常用的模式，可以使用更高效的算法或数据结构进行匹配。
* **避免重复扫描**: 实现一次遍历即可完成所有词法分析任务的策略，或者使用懒加载技术仅在必要时解析特定部分。
* **高效数据结构**: 根据具体需求选择合适的数据结构存储词汇表，如使用散列表加速查找，Trie树优化前缀匹配等。
* **并行处理**: 如果硬件条件允许，可以考虑将大型文件分割成多个部分，并行进行词法分析，最后合并结果。
* **预处理器优化**: 在正式的词法分析之前，可以先通过一个简单的预处理步骤去除无关的空白字符、注释等，减少后续处理的负担。

### (2) 实验体会

通过本次实验，我深刻理解了词法分析在程序设计中的重要性。

词法分析是编译器前端的重要组成部分，它负责将源代码中的字符序列转换为具有意义的单词符号，为后续的语法分析和语义分析提供基础。

通过本次实验，我对词法分析的原理和实现有了更深入的理解，并且学会了如何优化词法分析器以提高其效率。这对于我今后的程序设计和编译器开发工作将有很大的帮助。

## 八、参考资料

《编译原理课程实践任务书》

**实验三 LL（1）文法构造**

学生姓名：马星 学 号：5418122020 专业班级： 计算机科学与技术(卓越工程师)221班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期：2024/10/25 实验成绩：

## 一、实验项目名称

熟悉LL（1）文法的分析条件，了解LL（1）文法的构造方法。

## 二、实验目的

1、编制一个能够将一个非LL（1）文法转换为LL（1）文法；

2、消除左递归；

3、消除回溯。

## 三、实验基本原理

1. 将一个可转换非LL（1）文法转换为LL（1）文法，要经过两个阶段，1）消除文法左递归，2）提取左因子，消除回溯。
2. 消除文法左递归算法：

1）对文法G的所有非终结符进行排序

2）按上述顺序对每一个非终结符Pi依次执行:

for( j=1； j< i-1；j++)

将Pj代入Pi的产生式（若可代入的话）；

消除关于Pi的直接左递归：

Pi -> Piα|β ，其中β不以Pi开头，则修改产生式为：

Pi —＞ βPi′

Pi′—＞ αPi′|ε

3）化简上述所得文法。

1. 提取左因子的算法：

A —＞ δβ1|δβ2|…|δβn|γ1|γ2|…|γm

(其中,每个γ不以δ开头)

那么,可以把这些产生式改写成

A —＞ δA*′*|γ1| γ2…|γm

A*′*—＞β1|β2|…|βn

1. 利用上述算法，实现构造一个LL（1）文法：
   1. 从文本文件g.txt中读入文法，利用实验1的结果，存入实验1设计的数据结构；
   2. 设计函数remove\_left\_recursion（）和remove\_left\_gene（）实现消除左递归和提取左因子算法，分别对文法进行操作，消除文法中的左递归和提出左因子；
   3. 整理得到的新文法；
   4. 在一个新的文本文件newg.txt输出文法，文法输出按照一个非终结符号一行，开始符号引出的产生式写在第一行，同一个非终结符号的候选式用“|”分隔的方式输出。

## 四、主要仪器设备及耗材

PC微机

DOS操作系统或 Windows 操作系统

Turbo C 程序集成环境或 Visual C++ 程序集成环境

## 五、实验步骤（完整内容见光盘）

### 1、学习LL（1）文法的分析条件；

①非左递归：文法中不能包含直接或间接左递归，即形式如A → Aα | β的产生式是不允许的。

②FIRST集不相交：任意两个具有相同左部的产生式A→α|β，需要满足FIRST(α) ∩ FIRST(β) = ∅。

③ε限制：如果存在A→ε, 则需要满足FIRST(A)∩FOLLOW(A)= ∅。

这些条件确保了在分析过程中每一步选择都是确定且唯一的，从而使得语法分析过程无回溯。

### 2、学习构造LL（1）文法的算法；

①消除文法左递归

1）对文法G的所有非终结符进行排序

2）按上述顺序对每一个非终结符Pi依次执行:

for( j=1； j< i-1；j++)

将Pj代入Pi的产生式（若可代入的话）；

消除关于Pi的直接左递归：

Pi -> Piα|β ，其中β不以Pi开头，则修改产生式为：

Pi —＞ βPi′

Pi′—＞ αPi′|ε

3）化简上述所得文法。

②提取左因子，消除回溯。

A —＞ δβ1|δβ2|…|δβn|γ1|γ2|…|γm (其中,每个γ不以δ开头)

那么,可以把这些产生式改写成

A —＞ δA′|γ1| γ2…|γm

A′—＞β1|β2|…|βn

### 3、结合实验1给出的数据结构，编程实现构造LL（1）文法的算法；

#### (1) 第一部分, 消除直接左递归算法

/\*\*

\* P的产生式是否存在左递归

\* @param P 产生式左部的非终结符

\* @param rights 产生式右部

\*/

static bool haveLeftRecursion(const string &P, const vector<vector<string>> &rights) {

for (auto r: rights) {

if (r[0] == P)return true;// D -> D α 存在左递归

}

return false;

}

/\*\*

\* 消除直接左递归

\* D -> D α | β

\*

\* D —> β D′

\* D′ —> αP′|ε

\* @param P 非终结符, 对P->...消除左递归

\* @param source 全部产生式

\*/

void remove\_left\_recursion(const string &P, map<string, vector<vector<string>>> &source) {

if (!haveLeftRecursion(P, source[P]))return;

// D -> D α | β

vector<vector<string>> rights = source[P];

source[P].clear();

string PPrime = getNext(P);

for (auto &r: rights) {

if (r[0] == P) { // D -> D α 左递归式

vector<string> newR(r.begin() + 1, r.end());// α

newR.push\_back(PPrime);// αP′

source[PPrime].push\_back(newR); // D′ —> αP′|ε

} else {// D -> β 非左递归式

r.push\_back(PPrime);//βP′

source[P].push\_back(r); // D —> β D′

}

}

source[PPrime].push\_back({EMPTY}); // 空串

}

#### (2) 第二部分, 提取公共左因子算法

/\*\*

\* 以prefix为公共左因子, 对P的产生式进行合并

\* D —> δβ1|δβ2|...|δβn|γ1|γ2|...|γm

\*

\* D —> δP`|γ1|γ2|...|γm

\* D` —> β1|β2|...|βn

\* @param P 非终结符

\* @param source 产生式

\* @param prefix P的产生式的公共左因子δ

\*/

void merge(const string &P, map<string, vector<vector<string>>> &source, vector<string> &prefix) {

string PPrime = getNext(P);// D'

vector<vector<string>> rights = source[P];

source[P].clear();

bool haveE = false;// D -> δ@ β为空时P'需要生成空字

for (auto &right: rights) {

if (startsWith(right, prefix)) { // right含有公共左因子, 后面是β项, D' —> β1|β2|…|βn

if (prefix.size() != right.size()) { // β放入P'的产生式 D' -> β

source[PPrime].emplace\_back(right.begin() + prefix.size(), right.end());

} else if (!haveE) {// D' -> @

haveE = true;

source[PPrime].push\_back({EMPTY});

}

} else {// right不含公共左因子, 后面是γ项, D —> δP′|γ1| γ2…|γm

source[P].push\_back(right);

}

}

prefix.push\_back(PPrime);// 最后放入P'到P, D -> δP′

source[P].push\_back(prefix);

}

/\*\*

\* 对P的产生式进行公共左因子提取

\* D —> δβ1|δβ2|...|δβn|γ1|γ2|...|γm

\*

\* D —> δP`|γ1|γ2|...|γm

\* D` —> β1|β2|...|βn

\* @param P 非终结符

\* @param source 产生式

\* @return 是否有公共左因子

\*/

bool doLeft(const string &P, map<string, vector<vector<string>>> &source) {

vector<vector<vector<string>>> subsets = generateSubsetsAndSort(source[P]);// 生成子集(大小>1),从大到小排序

// 检查子集是否有左因子可以提取

for (auto &subset: subsets) {

// D —> δβ1|δβ2|…|δβn|γ1|γ2|…|γm

vector<string> prefix = commonPrefix(subset);//prefix=δ

if (prefix.empty()) continue;// 无公共前缀

// 可以提取

merge(P, source, prefix);

return true;

}

return false;

}

/\*\*

\* 对文法的全体产生式提取左因子(消除回溯)

\* @param source 产生式

\*/

void remove\_left\_gene(map<string, vector<vector<string>>> &source) {

bool change=true;

while (change) {

change=false;

for (auto &[Pi, v]: source) {

if(doLeft(Pi, source)){// 对每个产生式进行提取公共因子

change=true;// 是否提取了公共因子

}

}

}

}

#### (3) 第三部分 消除左递归和回溯

/\*\*

\* 消除左递归和回溯

\* @param source 原文法产生式

\* @return 消除左递归和回溯后的产生式

\*/

map<string, vector<vector<string>>> LL(vector<string> Vn, map<string, vector<vector<string>>> source) {

for (int i = 0; i < Vn.size(); i++) {

for (int j = 0; j < i; j++) {

PjIntoPi(Vn[j], Vn[i], source);

}

remove\_left\_recursion(Vn[i], source); // 消除Pi的直接左递归

}

remove\_left\_gene(source);// 提取左因子(消除回溯)

return source;

}

/\*\*

\* 是否存在Pi -> Pj α

\*/

bool PiHavePj(const string &Pi, const string &Pj, map<string, vector<vector<string>>> source) {

for (auto right: source[Pi]) {

if (right[0] == Pj) return true;

}

return false;

}

/\*\*

\* 如果 Pi -> Pj α, 则将Pj代入Pi的产生式

\*/

void PjIntoPi(const string &Pj, const string &Pi, map<string, vector<vector<string>>> &source) {

// 检查Pi的产生式的开头中是否有Pj

if (!PiHavePj(Pi, Pj, source))return;

vector<int> needDel;// Pi的第几个产生式需要被替换

vector<vector<string>> needAdd;// 需要替换为哪些产生式

for (int k = 0; k < source[Pi].size(); k++) {

vector<string> right = source[Pi][k];

if (right[0] != Pj)continue;

// 将Pj代入Pi的产生式

vector<vector<string>> res = replace(right, source[Pj]);

needDel.push\_back(k);

for (const vector<string> &r: res) needAdd.push\_back(r);

}

// 执行替换操作

modifyVector(source[Pi], needDel, needAdd);

}

### 4、结合实验1编程和调试实现对一个具体文法运用上述算法，构造它的LL（1）文法形式；

int main() {

Grammar g;

g.readFromFile("../Ex3/g3.txt");

cout << "源文件读取:" << g.sourceCode << endl;

LLFactor fac;

cout << "转为LL(1)文法:\n";

map<string, vector<vector<string>>> LL1 = fac.LL(g.getVn(), g.P);

print(LL1);

const char \*savePath = "../Ex3/out.txt";

saveToFile(LL1, savePath);

cout << "输出文件已保存在" << savePath << endl;

return 0;

}

### 5、把实验结果写入一个新建立的文本文件。

void saveToFile(map<string, vector<vector<string>>> source, const string &filename) {

ofstream outFile(filename);

if (!outFile) {

cerr << "无法打开文件: " << filename << endl;

return;

}

for (const auto &[k, v]: source) {

outFile << k << "->";

for (int i = 0; i < v.size(); i++) {

for (const string &j: v[i]) {

outFile << j;

}

if (i != v.size() - 1) {

outFile << "|";

}

}

outFile << ";\n";

}

outFile.close();

}

## 六、实验数据及处理结果

输入数据：

编辑一个文本文文件g.txt，在文件中输入如下内容：

S->Qc|c|cab;

Q->Rb|b;

R->Sa|a;

正确结果：

本实验的输出结果是不唯一的，根据消除左递归是选择非终结符号的顺序不同，或选择新的非终结符号的不同，可能会得到不同的结果，下面只是可能的一个结果：

S->Qc|cT;

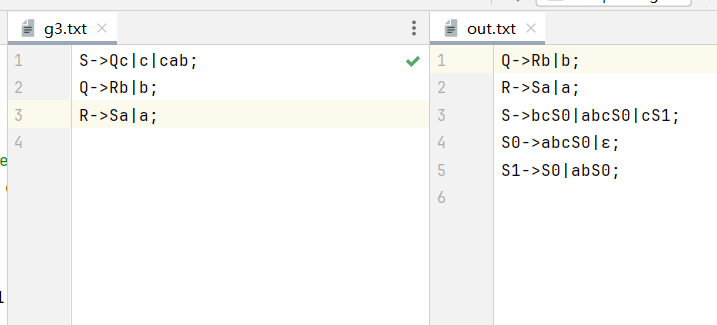
T->@|ab; //由于无法输出ε，用@代替

Q->Rb|b;

R->bcaU | cTaU | aU;

U->bcaU|@;

顺序采用字典序QRS



采用不同的顺序会得到不同的结果, 经手动验算, 程序的转换结果是正确的

## 七、思考讨论题或体会或对改进实验的建议

### (1) 思考题

**1、是不是所有的文法都可以通过上述程序构造LL（1）文法？**

并非一切非LL(1)文法都能利用提取左因子的方法将它改造为LL(1)文法

**2、LL（1）文法在整个语法分析中的作用？**

* **提供明确的解析策略**：LL(1)文法通过预测分析表提供了一种明确的自顶向下的解析策略，使得解析器能够根据当前输入符号和栈顶的非终结符唯一确定下一步的动作。这种明确的解析策略简化了编译器的实现，提高了解析效率。
* **保证解析的正确性和无二义性**：由于LL(1)文法消除了左递归和公因子，因此可以保证解析过程的正确性和无二义性。这对于编译器来说是非常重要的，因为任何解析错误都可能导致编译失败或生成错误的代码。
* **支持错误处理**：LL(1)文法的预测分析表还可以用于错误处理。当解析器遇到不符合文法规则的输入时，可以根据预测分析表给出明确的错误信息，帮助用户定位和修复错误。
* **提高编译器的性能**：由于LL(1)文法的解析过程是确定的，因此编译器可以在编译时生成高效的解析代码，从而提高编译器的性能。

### (2) 实验体会

通过本次实验，我深入理解了LL(1)文法的概念及其重要性。

学会了如何从一般的上下文无关文法转换为LL(1)文法，这包括消除左递归、提取公因子以及处理ε-产生式等关键技术步骤。

在尝试将非LL(1)文法转换为LL(1)文法的过程中，我遇到了一些挑战，如如何处理复杂的嵌套结构和多重选择的情况。通过查阅资料和反复试验，我逐渐找到了解决问题的方法。这一过程锻炼了我的问题解决能力和创新思维。

总的来说，本次实验不仅提高了我的专业技能，还增强了我对计算机科学基础概念的理解。我相信这些经验将在未来的学习和工作中发挥重要作用。未来，我希望能够进一步探索其他类型的文法和解析技术，以拓宽我的技术视野。

## 八、参考资料

《编译原理课程实践任务书》

**实验四 递归下降分析程序**

学生姓名：马星 学 号：5418122020 专业班级： 计算机科学与技术(卓越工程师)221班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期：2024/10/25 实验成绩：

## 一、实验项目名称

递归下降分析程序

## 二、实验目的

通过设计、编制、调试一个典型的语法分析程序，实现对词法分析程序所提供的单词序列进行语法检查和结构分析，进一步掌握常用的语法分析中递归下降分析方法。

设计一个文法的递归下降分析程序，判断特定表达式的正确性。

## 三、实验基本原理

1、 给出文法如下：

G[E]

E->T|E+T;

T->F|T\*F;

F->i|(E);

利用实验3的方法将上述文法转化为LL（1）文法；

2、 根据递归下降分析方法为转化后的文法设计递归下降分析程序，利用C语言或C++语言实现；

3、 利用递归下降分析程序完成下列功能：

1） 手工将测试的表达式写入文本文件，每个表达式写一行，用“；”表示结束；

2） 读入文本文件中的表达式；

3） 调用实验2中的词法分析程序搜索单词；

4） 把单词送入递归下降分析程序，判断表达式是否正确（是否是给出文法的语言），若错误，应给出错误信息；

5） 完成上述功能，有余力的同学可以对正确的表达式计算出结果。

## 四、主要仪器设备及耗材

PC微机

DOS操作系统或 Windows 操作系统

Turbo C 程序集成环境或 Visual C++ 程序集成环境

## 五、实验步骤（完整内容见光盘）

### 1、分析文法，将给出的文法转化为LL(1)文法；

使用实验三定义的LL(1)文法的转换程序, 将输入文法转换为LL(1)文法

### 2、学习递归下降分析程序的结构，设计合理的递归下降分析程序；

#### (1)递归下降子程序一般性伪代码

为了在不改变程序结构情况下, 实现对不同文法的识别, 我采用了生成式的递归下降, 只定义一个递归程序, 将非终结符作为参数传入, 用哈希映射做识别、递归

Procedure(vn):

if sym ∈ First[vn\_right]: // sym属于某个候选项的first集合

for ch in vn\_right:

if ch是非终结符:

Procedure (ch) // 递归下降

else: // 终结符

if sym==ch: // 匹配,读入下一个字符

advance()

else: // 不匹配, 报错

error()

else: // sym不在first集合中

if vn->ε&& sym∈Follow[vn]: // 候选项有空字时, sym属于follow集合

// 不报错

else:

error() // 报错

#### (2)构建firs集合

①A → aβ 则 a∈First(A)

②A → ε 则 ε∈First(A)

③A → B1B2...Bk

​ 如果B1到Bi都可以推出ε, 则j∈[1,i], First(Bj) - ε ∈First(A)

​ 如果∀i都有Bi→ ε, 则ε ∈First(A)

由于计算First(A)时可能使用未计算的First(B), 所以需要重复进行扫描, 直到完整扫描一遍无法再扩展任一First集

/\*\*

\* 根据文法创建first集合

\* @param source 文法的产生式

\* @return first集合

\*/

static map<string, set<string>> createFirst(

const map<string, vector<vector<string>>> &source) {

map<string, set<string>> first;

bool isModify = true;

while (isModify) {

isModify = false;

for (const auto &[P, rights]: source) {// Production -> [right1|...]

set<string> &firstP = first[P];

int originalSize = firstP.size();

for (const auto &right: rights) {

const set<string> &fp = selectFirst(first, source, right);

firstP.insert(fp.begin(), fp.end());

}

if (firstP.size() != originalSize)isModify = true;

}

}

return first;

}

/\*\*

\* 查询P->right的First集合

\* @param first 已计算出的First集合

\* @param source 文法产生式集合

\* @param right 一个候选项

\*/

static set<string> selectFirst(map<string, set<string>> &first,

const map<string, vector<vector<string>>> &source,

const vector<string> &right) {

set<string> ans;

for (const auto &r: right) {

if (isVn(r)) {// 非终结符

set<string> fr = first[r];

if (haveEmpty(source.at(r), EMPTY)) {// r->@

fr.erase(EMPTY);

ans.insert(fr.begin(), fr.end());

} else {// 不含@

ans.insert(fr.begin(), fr.end());

break;

}

} else {

ans.insert(r);

break;

}

}

return ans;

}

#### (3)构建follow集合

①文法开始符号: # ∈ Follow(S)

②A→αXβ 则 First(β) - ε∈ Follow(X)

③A→αX 则 Follow(A) ∈ Follow(X)

④A→αXβ ε∈First(β) 则 Follow(A) ∈ Follow(X)

/\*\*

\* 根据文法和first集合创建follow集合

\* @param start 文法的开始符号

\* @param first 文法的first集合

\* @param source 文法的产生式

\* @return follow集合

\*/

static map<string, set<string>> createFollow(

const string &start, const map<string, set<string>> &first,

const map<string, vector<vector<string>>> &source) {

map<string, set<string>> follow;

follow[start].insert(END);// 开始符号的follow集合放入END

bool isModify = true; // 是否还有要计算的follow

while (isModify) {

isModify = false;

for (const auto &[P, rights]: source) {

for (auto right: rights) { // 遍历每个候选项, 查找非终结符进行处理

dealRight(first, source, P, right, follow, isModify);

}

}

}

return follow;

}

/\*\*

\* 处理P->right的候选项follow集合

\* @param first 计算好的first集合

\* @param source 文法产生式

\* @param P 非终结符, P->right

\* @param right 候选项, P->right

\* @param follow 当前计算出的follow集合

\* @param isModify 该方法是否修改了follow集合

\*/

static void dealRight(const map<string, set<string>> &first,

const map<string, vector<vector<string>>> &source,

const string &P,

const vector<string> &right,

map<string, set<string>> &follow,

bool &isModify) {

// 正序遍历, 若pre为非终结符, 则 first[suf]- ε ∈ follow[pre]

for (int i = 0; i < right.size() - 1; i++) {

string pre = right[i], suf = right[i + 1];

if (!isVn(pre))continue;

if (isVn(suf)) {// suf为非终结符, 将first(suf)-ε加入follow(pre)

set<string> firstSuf = first.at(suf);

firstSuf.erase(EMPTY);

if (merge(follow[pre], firstSuf))isModify = true;

} else {// suf为终结符, 将suf加入follow(pre)

if (merge(follow[pre], suf))isModify = true;

}

}

string last = right.back();// 最后一个字符没有处理到

if (isVn(last)) {

if (merge(follow[last], follow[P]))isModify = true;

}

// 倒序遍历, 若pre后面均能推出空字, 则follow[P] ∈ follow[pre]

for (int i = right.size() - 2; i >= 0; i--) {

string pre = right[i], suf = right[i + 1];

if (isVn(suf) && haveEmpty(source.at(suf), EMPTY)) {

if (isVn(pre)) {

if (merge(follow[pre], follow[P]))isModify = true;

}

} else {

break;

}

}

}

#### (4)递归下降子程序

int point = 0;// 字符串指针

vector<Token> expression;// 正在判断的输入串

string sym;//当前字符

bool isRightExpression = true;

void ERROR() {

isRightExpression = false;

}

void advance() {

Token &token = expression[point++];

if (token.type == DecInt || token.type == Identifier)sym = "i";

else sym = token.value;

}

bool firstContains(vector<string> right, string sym) {

string r = right[0];

if (isVn(r)) {

return contains(first[r], sym);

} else {

return r == sym;

}

}

void P(const string &p) {

vector<vector<string>> &rights = source[p];

bool match = false;

for (const auto &right: rights) {

// 查找first集合中匹配sym的候选项

if (!firstContains(right, sym)) continue;

match = true;

for (const string &r: right) {// 非终结符:执行对应的程序; 终结符:匹配后执行advance

if (isVn(r)) {// r为非终结符

P(r);// 递归下降

} else {// r为终结符

if (sym == r)advance();// 匹配

else ERROR();// 无法匹配

}

}

break;

}

if (!match) {

if (!haveEmpty(rights, EMPTY))ERROR();// 无法推导出空字

else if (!contains(follow[p], sym))ERROR();// 可以推导出空字, 但follow仍无法匹配

// else 不报错

}

}

#### (5)判断表达式合法性的方法

/\*\*

\* 判断表达式是否合法

\*/

bool judge(const vector<Token> &e) {

this->expression = e;

point = 0;

isRightExpression = true;

advance();

P(start);

return isRightExpression && sym == END;

}

### 3、编写测试程序，包括表达式的读入和结果的输出；

int main() {

// 调用实验一定义的文法输入输出结构对输入文法进行处理

Grammar g;

g.readFromFile(R"(E:\CLionProject\C++Projects\Analysis\Ex4\g4.txt)");

cout << "源文件读取:\n" << g.sourceCode << endl;

// 调用实验三定义的LL(1)文法转换程序进行处理

LLFactor fac;

cout << "转为LL(1)文法:\n";

map<string, vector<vector<string>>> source = fac.LL(g.getVn(), g.P);

print(source);

// 预处理first和follow集合

cout << "递归下降程序预处理:\n";

Recursive r(g.S, source);

cout << "first集合:\n";

print(r.first);

cout << "follow集合:\n";

print(r.follow);

// 表达式输入

cout << "判断表达式:\n";

string expressions = read(R"(E:\CLionProject\C++Projects\Analysis\Ex4\expressions.txt)");

istringstream stream(expressions);

string expression;

WordFactor wordFactor;// 实验二定义的词法分析器

while (getline(stream, expression)) {

expression.pop\_back();

vector<Token> tokens = wordFactor.scan(expression);// 词法分析

tokens.emplace\_back(Keyword, END);

bool isRight = r.judge(tokens); // 用递归下降子程序进行判断

cout << setw(18) << left << expression << (isRight ? "正确" : "不正确") << endl;

}

}

### 4、测试程序运行效果，测试数据可以参考下列给出的数据。

输入数据:

10;

1+2;

(1+2)\*3+(5+6\*7);

((1+2)\*3+4;

1+2+3+(\*4+5);

(a+b)\*(c+d);

((ab3+de4)\*\*5)+1;

正确结果：

（1）10;

输出：正确

（2）1+2;

输出：正确

（3）(1+2)\*3+(5+6\*7);

输出：正确

（4）((1+2)\*3+4

输出：错误

（5）1+2+3+(\*4+5)

输出：错误

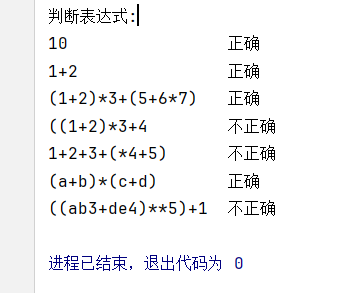
（6）(a+b)\*(c+d)

输出：正确

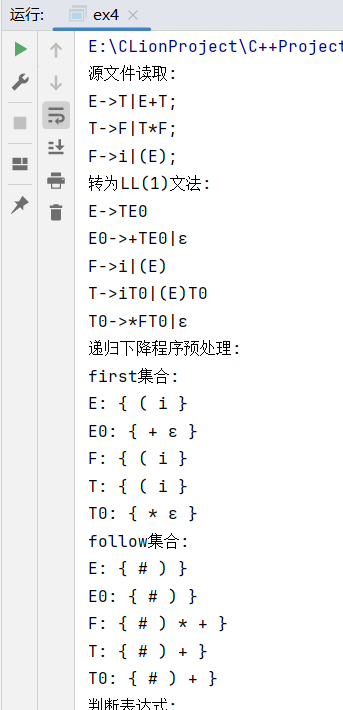
（7）((ab3+de4)\*\*5)+1

输出：错误

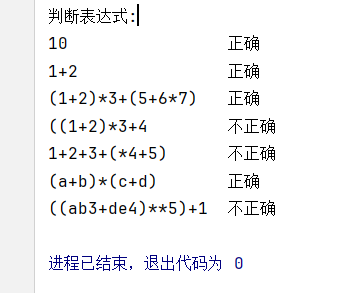
程序运行结果:



LL(1)文法以及first、follow集合输出如下:



## 六、实验数据及处理结果



实验结果与预期一致

## 七、思考讨论题或体会或对改进实验的建议

### (1) 思考题

1. **为什么文法必须先转化为LL（1）文法再来做递归下降分析？**

① 如果文法存在左递归, 可能导致递归下降分析进行死循环

② 如果文法存在回溯, 则递归下降分析需要对每个可能匹配的分支都进行遍历和回退, 效率很低，代价极高

1. **如果用预测分析法来改写本程序，如何来实现？**

先根据first和follow集合构建预测分析表

栈内放入文法开始符号

扫描输入串, 对于栈顶符号X,和当前输入符号a:

①若X=a=”#”, 则表达式正确

②若X为终结符, X=a≠”#”,则X与a匹配, X出栈,a读入下一个字符

③若X为非终结符, 则查询预测分析表

若表中[X,a]有一个产生式, 则X出栈, 产生式从右到左入栈

若表中[X,a]无产生式, 则报错

### (2) 实验体会

1. **理解递归下降分析方法**：
   * 在实验中，我深入理解了递归下降分析的基本概念和实现方法。通过将文法转化为LL(1)文法，再使用递归下降分析方法，能够有效地对输入表达式进行语法检查。
   * 递归下降分析的核心思想是利用文法的产生式规则，从左到右逐步解析输入串。当遇到非终结符时，继续递归调用相应的子程序；当遇到终结符时，根据当前状态决定是匹配还是报错。
2. **处理复杂表达式的能力提升**：
   * 通过编写和调试递归下降分析程序，我学会了如何处理复杂的表达式，包括嵌套的括号、多种运算符（如加减乘除）以及不同优先级的运算。
   * 在实现过程中，我特别注意了运算符的优先级和结合性问题。通过引入中间变量和调整递归顺序，确保了运算符的正确计算顺序。
3. **调试与测试的重要性**：
   * 在实验中，调试和测试是不可或缺的环节。通过不断地测试不同的输入表达式，我发现了程序中的许多潜在错误，并进行了修正。
   * 测试不仅帮助我验证了程序的正确性，还让我更加熟悉了递归下降分析的原理和应用。
4. **代码优化与改进**：
   * 在实验过程中，我不断尝试优化代码结构，提高程序的可读性和可维护性。例如，通过引入辅助函数和合理的注释，使得代码更加清晰易懂。
   * 我还学习了如何利用调试工具进行代码跟踪和性能分析，以便发现和解决程序中的问题。
5. **对编译原理课程的理解加深**：
   * 通过这次实验，我对编译原理课程的内容有了更深刻的理解。特别是对语法分析部分，我掌握了从文法定义到程序实现的全过程。
   * 实验不仅提高了我的编程技能，还增强了我对计算机科学基础概念的理解和应用能力。

这次实验让我深刻体会到了递归下降分析方法在语法分析中的重要性和应用价值。通过实践操作，我不仅掌握了相关技术，还培养了解决问题的能力和创新思维。我相信这些经验将在未来的学习和工作中发挥重要作用。

## 八、参考资料

《编译原理课程实践任务书》

**实验五 逆波兰式的翻译和计算**

学生姓名：马星 学 号：5418122020 专业班级： 计算机科学与技术(卓越工程师)221班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期：2024/11/22 实验成绩：

## 一、实验项目名称

逆波兰式的翻译和计算

## 二、实验目的

通过实验加深对语法指导翻译原理的理解，掌握LR分析的方法，将语法分析所识别的表达式变换成中间代码的翻译方法。

设计一个表示能把普通表达式（中缀式）翻译成后缀式，并计算出结果的程序。

## 三、实验基本原理

1、给出文法如下：

G[E]

E->T|E+T;

T->F|T\*F;

F->i| (E);

对应的转化为逆波兰式的语义动作如下：

E-> E(1)op E(2) {E.CODE:= E(1).CODE||E(2).CODE||op}

E->(E(1)) { E.CODE := E(1).CODE}

E->id { E.CODE := id}

2、利用LR分析算法，结合上面给出的语义动作实现逆波兰式的构造；

3、利用栈，计算生成的逆波兰式，步骤如下：

1） 中缀表达式，从文本文件读入，每一行存放一个表达式，为了降低难度，表达式采用常数表达式；

2） 利用结合语法制导翻译的LR分析，构造逆波兰式；

3） 利用栈计算出后缀式的结果，并输出；

## 四、主要仪器设备及耗材

PC微机

DOS操作系统或 Windows 操作系统

Turbo C 程序集成环境或 Visual C++ 程序集成环境

## 五、实验步骤（完整内容见光盘）

1、了解语法制导翻译的方法，学习后缀式构造的语义动作；

2、结合LR分析，设计程序构造后缀式；

3、利用栈，编程实现后缀式的计算；

4、测试程序运行效果：从文本文件中读表达式，在屏幕上输出，检查输出结果。

(由于总想着做通用化, 能够识别各种文法, 导致这个实验过于复杂了)

### (1) 拓广文法

在实验一的文法基础上加入新的开始符号S’, S’->S

并对文法的产生式进行标号, 不再采用哈希表存储

class ExtendGrammar {// 拓广文法

public:

string originStart;// 原文法开始符号

string S;// 拓广文法开始符

vector<Production> P;// 拓广文法产生式

set<string> Vn, Vt;// 终结符与非终结符

Production startProject;// 拓广文法的开始产生式

Generator pOrderGen;// 产生式标号生成器

explicit ExtendGrammar(const Grammar &grammar) {

originStart = grammar.S;

Vn = grammar.Vn;

Vt = grammar.Vt;

// 拓广文法

S = originStart + "'";

Vn.insert(S);

startProject = {pOrderGen.get(), S, vector{originStart}};

P.push\_back(startProject);

for (const auto &[left, rights]: grammar.P) {

for (const auto &right: rights) {

P.emplace\_back(pOrderGen.get(), left, right);

}

}

}

/\*\*

\* 获取第i个文法

\* @param i 索引

\*/

Production order(int i) const {

return P[i];

}

/\*\*

\* 获取产生式的标号

\* @param find 要查找的产生式

\* @return 标号, 未找到返回-1

\*/

int getOrder(const Production& find) const {

for (auto p: P) {

if (p.left == find.left && p.right == find.right) {

return p.order;

}

}

return -1;

}

};

### (2) 构建能识别活前缀的DFA

① 将文法的开始项目全部计入I0

② 从I0开始扫描项目集中的项目,

若Ik中存在A->α·xβ, 则Im->In存在一条x边, x可以为终结符或非终结符. 对于归约项目A->α·不做处理

③ 当In被创建时, 若存在A->α·Xβ, 则需要做闭包, 将X的开始项目全部计入In(递归性的)

④ 对于拓广文法S`->S,存在S`->S·的项目集为结束项目集(创建action-goto表时写入acc)

class DfaFactor {// 拓广文法 -> 项目集规范簇DFA

public:

map<int, I \*> id\_I\_map;// id -> 项目集规范簇

set<I \*> visitedI;// 已计算过的项目集规范簇

map<I \*, map<string, I \*>> dfa;// I -> 边 -> I

Generator I\_idGen;// 项目集id生成器

explicit DfaFactor(const ExtendGrammar &extendGrammar) {

// 建立项目集规范簇

createDFA(extendGrammar);

}

/\*\*

\* 根据拓广文法构建DFA项目集规范簇

\*/

void createDFA(const ExtendGrammar &extendGrammar) {

I \*I0 = new I(I\_idGen.get());

id\_I\_map[I0->id] = I0;

// 把S'->·S放入I0

Production startPro = extendGrammar.startProject;

startPro.addFirst(DOT);

I0->insert(startPro);

I0->merge(Closure(DOT, extendGrammar.P, extendGrammar.originStart));// S做闭包

// Goto

queue<I \*> q;

q.push(I0);

while (!q.empty()) {

I \*i = q.front();

q.pop();

if (visitedI.find(i) != visitedI.end())continue;

visitedI.insert(i);

set<string> dotNext = i->getDotNext(DOT);// 可以走的边

//cout << \*i;

for (const string &edgeChar: dotNext) {

I \*newI = i->Goto(DOT, extendGrammar.P, edgeChar);

I \*exist = findExist(newI);

if (exist != nullptr) {// 已存在的项目集 -> 返回已存在的项目集

newI = exist;

} else {// 新项目集, 分配id

newI->id = I\_idGen.get();

id\_I\_map[newI->id] = newI;

q.push(newI);

}

dfa[i][edgeChar] = newI;

//cout<<"走"<<edgeChar<<"连接I"<<newI->id<<endl;

}

}

}

I \*findExist(I \*newI) {

for (auto &[id, i]: id\_I\_map) if (\*i == \*newI) return i;

return nullptr;

}

};

### (3) 构建Action-Goto表, 使用SLR(1)

① 若Im->In的边为x, 则Action[Sm,x]=Sn

② 若Im->In的边为X, 则goto[Sm,X]=Sn

③ 若Im中存在归约项目, [k] A->α•, 则Action[Sm,f]=rk, 其中f∈Follow(A)

④ 若Im中的归约项目与移进项目仍然发生冲突, 则报错并终止程序

class SheetFactor {// DFA -> Action-Goto表

public:

using Operator = pair<string, int>;// 移进Sx | 归约rx

map<int, map<string, Operator>> ActionSheet;// 状态 + 终结符 -> 操作(sx|rx)

map<int, map<string, int>> GotoSheet;// 状态 + 非终结符 -> 生成状态

set<string> Vn, Vt;// 非终结符、终结符

map<I \*, map<string, I \*>> dfa;// 能识别活前缀的DFA

map<int, I \*> id\_I\_map;// 项目集id -> 项目集

map<string, set<string>> Follow;

SheetFactor(const ExtendGrammar &extendGrammar,

const map<I \*, map<string, I \*>> &dfa,

const map<int, I \*> &id\_I\_map,

const map<string, set<string>> &Follow) {

Vn = extendGrammar.Vn;

Vt = extendGrammar.Vt;

this->dfa = dfa;

this->id\_I\_map = id\_I\_map;

this->Follow = Follow;

dfaToSheet(extendGrammar);

}

/\*\*

\* 通过DFA建立分析表

\* @param extendGrammar 拓广文法

\*/

void dfaToSheet(const ExtendGrammar &extendGrammar) {

Vt.insert(END);

Production acceptP{ // 接受项目

extendGrammar.startProject.order,

extendGrammar.S,

{extendGrammar.originStart, DOT}

};

// 待约|移进项目

for (auto &[id, items]: id\_I\_map) {

map<string, I \*> edges = dfa.at(items);

for (const auto &[edgeChar, nextI]: edges) {

if (isVn(edgeChar)) {

GotoSheet[id][edgeChar] = nextI->id;

} else {

ActionSheet[id][edgeChar] = {"S", nextI->id};

}

}

// 接受项目

if (items->contains(acceptP)) {

ActionSheet[id][END] = {"acc", 0};

}

// 归约项目

for (const auto &p: items->productions) {

auto right = p.right;

if (p.isReductionProject(DOT) && p != acceptP) {// 归约项目

// √ - SLR(1): ActionSheet[id][Follow] = rx

for (const string &follow: Follow[p.left]) {

ActionSheet[id][follow] = {"r", p.order};

}

}

}

}

}

};

### (4) 语义动作读入和预处理

将带下标的文法产生式及其语义动作读入, 进行解析和预处理, 方便原产生式与语义动作形成联系(写的时候没发现, 回头再来看这一块我可能想复杂了)

class Semantic {// 语义

public:

ExtendGrammar \*extendGrammar;

vector<Production> P;// 原文法产生式

vector<Production> semanticP;// 带下标的文法产生式

vector<vector<string>> action;// 语义动作

Semantic(ExtendGrammar &extendGrammar) {

this->extendGrammar = &extendGrammar;

this->P = extendGrammar.P;

semanticP.reserve(P.size());

semanticP.resize(P.size());

action.reserve(P.size());

action.resize(P.size());

}

void readFromFile(const string &filename) {// 读取输入文本

string sourceCode = read(filename);

vector<vector<string>> result = parse(sourceCode);// 解析语义动作文本

for (auto &line: result) {// line = [带标号的产生式,语义动作1,语义动作2,...]

string production = line[0];

Production sp = parseProduction(production);// 带标号的产生式

Production p = parseProduction(removeVnNumbers(production));// 不带标号的产生式

sp.order = extendGrammar->getOrder(p);// 通过不带标号的产生式查找拓广文法中产生式序号

semanticP[sp.order] = sp; // 产生式序号 -> 产生式|语义

for (int i = 1; i < line.size(); i++) {

action[sp.order].push\_back(line[i]);

}

}

}

/\*\*

\* 解析产生式-语义动作的输入文本

\* @param grammarSemantic 产生式-语义动作,

\* 格式为"带标号的产生式,语义动作1,语义动作2,...\n...",

\* 例如"F->i,F.code=i.val,F.val=i.val"

\* @return [[带标号的产生式,语义动作1,语义动作2,...],[...],...]

\*/

vector<vector<string>> parse(const string &grammarSemantic) {

vector<vector<string>> result;

istringstream stream(grammarSemantic);

string line;

while (getline(stream, line)) {

istringstream lineStream(line);

string part;

vector<string> lineParts;

while (getline(lineStream, part, ',')) {

lineParts.push\_back(part);

}

result.push\_back(lineParts);

}

return result;

}

static string removeVnNumbers(const string &s) {

regex pattern("([A-Z])\\d+"); // 匹配大写字母后跟的数字

return regex\_replace(s, pattern, "$1"); // 使用正则表达式替换匹配的部分为空字符串

}

/\*\*

\* 产生式文本解析为产生式对象

\*/

Production parseProduction(string production) {

int pos = production.find("->");

string left = production.substr(0, pos);

string r = production.substr(pos + 2);

vector<string> right = splitRightPart(r);

return Production(left, right);

}

/\*\*

\* 解析产生式右部

\*/

vector<string> splitRightPart(const string &right) {// E1+T ---> E1,+,T

vector<string> result;

string temp;

for (char ch: right) {

if (isalpha(ch)) {

if (!temp.empty()) {

result.push\_back(temp);

temp.clear();

}

temp += ch;

} else if (isdigit(ch)) {

temp += ch;

} else {

if (!temp.empty()) {

result.push\_back(temp);

temp.clear();

}

result.push\_back(string(1, ch));

}

}

if (!temp.empty()) result.push\_back(temp);

return result;

}

/\*\*

\* 对语义动作进行解析

\* @param input 语义动作

\* @return [ < 符号, 属性名 > ]

\*/

vector<pair<string, string>> analysisAction(const string &input) {

vector<pair<string, string>> result;

regex pattern(R"((\w+)\.(\w+))");

smatch match;

string::const\_iterator searchStart(input.cbegin());

while (regex\_search(searchStart, input.cend(), match, pattern)) {

result.emplace\_back(match[1], match[2]);

searchStart = match.suffix().first;

}

return result;

}

/\*\*

\* 参数替换

\* @param a 原字符串

\* @param args 替换参数

\* @return a="a||b||c" args=[1,2,3] -> "1||2||3"

\*/

string replace\_with\_args(const string &a, const vector<string> &args) {

stringstream ss(a);

string segment;

vector<string> segments;

while (getline(ss, segment, '|')) {

if (!segment.empty()) segments.push\_back(segment);

}

string result;

for (size\_t i = 0; i < segments.size(); ++i) {

if (i > 0) result += "||";

if (i < args.size()) result += args[i];

else result += segments[i];

}

return result;

}

/\*\*

\* 计算属性

\* @param a 语义动作

\* @param args 右侧参数

\* @return 计算结果

\*/

string calAction(string a, vector<string> args) {

int idx = a.find('=');

a = a.substr(idx + 1); // 例E0.code=E1.code||T.code||+ ---> E1.code||T.code||+

if (a.find("||") != string::npos) {// 连接(连接符一并返回)

return replace\_with\_args(a, args);// 例E1.code||T.code||+ ---> 1||2||+

}

// +或\*

if (a.find('+') != string::npos) {// 例E.val+T.val ---> 1+2=3

int sum = 0;

for (auto i: args)sum += stoi(i);

return to\_string(sum);

}

if (a.find('\*') != string::npos) {// 例T.val\*F.val ---> 3\*3=9

int sum = 1;

for (auto i: args)sum \*= stoi(i);

return to\_string(sum);

}

//赋值 例 F.val = i.val ---> i.val

return args[0];

}

};

### (5) 输入表达式, 自底向上构建语法树

通过之前构建好的Action-Goto表, 使用栈进行移进-归约, 并多加一个树节点栈, 与状态栈做相同的操作, 在移进时创建相应节点, 在归约时将要移进的父节点与要弹出的子节点相连, 最后返回语法树的根节点

我准备一步一步来, 所以此处的构建未做语义动作, 将构建过程与属性计算分离了

class GrammarTreeFactor {// 基于Action-Goto表对输入串归约构建语法树

public:

using Operator = pair<string, int>;// 移进Sx | 归约rx | acc

map<int, map<string, Operator>> Action;// 状态 + 终结符 -> 操作(sx|rx)

map<int, map<string, int>> Goto;// 状态 + 非终结符 -> 生成状态

GrammarTreeFactor(const map<int, map<string, Operator>> &ActionSheet,

const map<int, map<string, int>> &GotoSheet) {

this->Action = ActionSheet;

this->Goto = GotoSheet;

}

/\*\*

\* 自底向上建树

\* @param extendGrammar 拓广文法

\* @param tokens 输入串

\* @return 根节点, 如果失败返回nullptr

\*/

TreeNode \*creatTree(const ExtendGrammar &extendGrammar, vector<Token> tokens) {

stack<TreeNode \*> tree;// 节点栈, 用于建树

stack<int> status; // 状态栈

stack<string> chars; // 字符栈

stack<Token> input; // 输入串(统一格式,用栈遍历输入串)

// 初始入栈

tokens.emplace\_back(Keyword, END);

status.push(0);

chars.push(END);

for (int i = tokens.size() - 1; i >= 0; i--) input.push(tokens[i]);

//cout << "状态栈" << "\t\t\t" << "符号栈" << "\t\t\t" << "输入串" << "\t\t "

// << "操作" << "\t\t" << "说明" << endl;

string preOperator = "准备"; // 打印信息时使用

string preRemark;

while (true) {// 移进-归约

//printStk(status, true);

//printStk(chars, true);

//printStk(input, false);

int S\_top = status.top();

string char\_top = chars.top();

Token &nextToken = input.top();

string b = nextToken.type == DecInt ? "i" : nextToken.value;// 数字用i替换

map<string, Operator> &actionMap = Action[S\_top];

if (actionMap.find(b) == actionMap.end()) {

cout << "错误\t\t" << "Action[S" << S\_top << "," << b << "]=err" << endl;

return nullptr;

}

auto [type, num] = actionMap[b];

//cout << preOperator << "\t\t" << preRemark;

if (preOperator == "接受") return tree.top();

if (type == "acc") {

preOperator = "接受";

preRemark = "";

} else if (type == "S") {// 移进

preOperator = "移进";

preRemark = "";

status.push(num);

chars.push(b);

input.pop();

TreeNode \*pNode = new TreeNode(b);

if (nextToken.type == DecInt) {// i计入属性

pNode->attributes["val"] = nextToken.value;

pNode->attributes["code"] = nextToken.value;

}

tree.push(pNode);

} else {// 归约

preOperator = "归约";

Production p = extendGrammar.order(num);// 使用的产生式为第num个

preRemark = p.toString();

int cnt = p.right.size();// 出栈个数

stack<TreeNode \*> children;

for (int j = 0; j < cnt; j++) {

status.pop();

chars.pop();

children.push(tree.top());// 节点栈出栈的为子节点, 即将入栈的为父节点

tree.pop();

}

int &top = status.top();

map<string, int> &gotoMap = Goto[top];

if (gotoMap.find(p.left) == gotoMap.end()) { // Goto[S\_top,Vn]->S\_next

cout << "错误\t\t" << "Goto[S" << top << "," << p.left << "]=err" << endl;

return nullptr;

}

status.push(gotoMap[p.left]);

chars.push(p.left);

TreeNode \*pNode = new TreeNode(p.left);

pNode->setChildren(children);

pNode->useProductionId = num;

tree.push(pNode); // 生成父节点

}

//cout << endl;

}

}

};

### (6) 计算属性

由于属性全部是综合属性, 所以只需要按后序遍历, 自下而上的计算综合属性即可

另外, 我新增了一个val属性, 在翻译code的同时计算值val, 省去了用”栈计算翻译出的逆波兰表达式”这一步

class CalAttribute {

public:

Semantic \*semantic;

ExtendGrammar \*extendGrammar;

CalAttribute(Semantic &semantic, ExtendGrammar &extendGrammar) {

this->semantic = &semantic;

this->extendGrammar = &extendGrammar;

}

/\*\*

\* 树遍历法计算属性

\* @param node 当前遍历的节点

\*/

void visitNode(TreeNode \*node) {

if (node == nullptr)return;

cal(node); // 计算继承属性

for (auto &child: node->children) {

visitNode(child);// 遍历子树

}

cal(node); //计算综合属性

}

/\*\*

\* 计算属性

\* @param node 要计算的节点

\*/

void cal(TreeNode \*node) {

if (!isVn(node->name))return;// 终结符属性不需要计算

int id = node->useProductionId;

Production useProduction = extendGrammar->order(id);// node->children使用的产生式

vector<string> &action = semantic->action[id];// node->children使用的语义动作

for (auto &a: action) {

// 解析语义动作

vector<pair<string, string>> analysisAction = semantic->analysisAction(a);

// 属性是否计算过了

string &attributeName = analysisAction[0].second;

if (isCaled(node, attributeName)) continue;

// 确认属性能否计算

vector<string> \*args = canCal(node, analysisAction);

if (args == nullptr) continue;

// 计算属性结果

string result = semantic->calAction(a, \*args);

node->attributes[attributeName] = result;

}

}

/\*\*

\* node的attributeName是否计算过了

\*/

bool isCaled(TreeNode \*node, const string &attributeName) {

return node->attributes.find(attributeName) != node->attributes.end();

}

/\*\*

\* node的语义动作是否能执行

\* @return 如果能执行, 返回语义动作右式的参数列表, 否则返回nullptr

\*/

vector<string> \*canCal(TreeNode \*node, vector<pair<string, string>> analysisAction) {

vector<string> \*ans = new vector<string>();

for (int i = 1; i < analysisAction.size(); i++) {// 语义动作右侧

// 需要的节点和属性

string nodeName = analysisAction[i].first;

string attributeName = analysisAction[i].second;

TreeNode \*child = getNode(node, nodeName);// 根据节点名获取子节点

// 子节点的该属性是否存在

if (child->attributes.find(attributeName) == child->attributes.end()) {

return nullptr;

}

// 可以计算, 放入参数列表

ans->push\_back(child->attributes[attributeName]);

}

return ans;

}

/\*\*

\* 根据节点名获取子节点

\* @param node 父节点

\* @param nodeName 子节点名和下标

\* @return 例如 E0.code=E2.code||E1.code

\* 输入为E2会返回node子节点中名为E的第二个节点

\*/

TreeNode \*getNode(TreeNode \*node, string nodeName) {

int order = 1;// 是同名节点中的第几个

if (nodeName.size() > 1) {// nodeName带标号

order = stoi(nodeName.substr(1));

nodeName = nodeName.substr(0, 1);

if (node->name != nodeName)order++;// 如果与父节点名相同, 次序相对-1

}

// 查找子节点的第order个

for (auto &child: node->children) {

if (child->name == nodeName) {

order--;

if (order == 0) return child;

}

}

return nullptr;

}

};

### (7) 主函数测试

文法读入->拓广文法->识别活前缀的DFA->Action-goto表->

输入串->转为token流->构建语法树->属性计算->输出结果

/\*\*

\* 翻译并计算逆波兰表达式

\* @param expression 中缀式

\* @param extendGrammar 中缀式的拓广文法

\* @param sheetFactor Action-Goto表

\* @param wordFactor 词法分析,输入串变为Token流

\* @param calAttribute 树遍历法计算属性

\*/

void tranAndCal(const string &expression,

const ExtendGrammar &extendGrammar,

const SheetFactor &sheetFactor,

WordFactor &wordFactor,

CalAttribute &calAttribute) {

vector<Token> tokens = wordFactor.scan(expression);

// 自底向上构建语法树

GrammarTreeFactor grammarTreeFactor(sheetFactor.ActionSheet,

sheetFactor.GotoSheet);

TreeNode \*tree = grammarTreeFactor.creatTree(extendGrammar, tokens);

if (tree == nullptr) {

cout << "语法树构建失败" << endl;

return;

}

// 树遍历法计算属性

calAttribute.visitNode(tree);

// 输出code属性和value属性

cout << setw(30) << left << replaceAll(tree->attributes["code"], "||", ",");

cout << tree->attributes["val"] << endl;

}

int main() {

// 文法读入

Grammar g;

g.readFromFile("../Ex5/g5\_grammar.txt");

cout << "源文件读取:\n" << g.sourceCode << endl;

// SLR(1)需要计算follow集

map<string, set<string>> first = Recursive::createFirst(g.P);

map<string, set<string>> follow = Recursive::createFollow(g.S, first, g.P);

// 拓广文法 - 项目集规范簇DFA - 分析表

ExtendGrammar extendGrammar(g);

extendGrammar.print();

DfaFactor dfaFactor(extendGrammar);

dfaFactor.printDfaEdge();

SheetFactor sheetFactor(extendGrammar, dfaFactor.dfa, dfaFactor.id\_I\_map, follow);

sheetFactor.printSheet();

// 语义动作读入

Semantic semantic(extendGrammar);

semantic.readFromFile("../Ex5/g5\_semantic.txt");

// 准备翻译并计算逆波兰表达式

WordFactor wordFactor;// 词法分析器

CalAttribute calAttribute(semantic, extendGrammar);// 属性计算器

cout << "翻译并计算逆波兰表达式:\n";

string expressions = read("../Ex5/g5\_expression.txt");

istringstream stream(expressions);

string expression;

cout << setw(5) << left << "" << setw(30) << left << "input"

<< setw(30) << left << "translate" << "calculate" << endl;

int i = 0;// 第几个输入

while (getline(stream, expression)) {

i++;

expression.pop\_back();

cout << setw(5) << left << ("[" + to\_string(i) + "]");

cout << setw(30) << left << expression;

tranAndCal(expression, extendGrammar, sheetFactor, wordFactor, calAttribute);

}

return 0;

}

### (8) 程序运行结果

输入数据:

1+2;

(1+2)\*3;

(10+20)\*30+(50+60\*70);

正确结果：

（1）1+2;

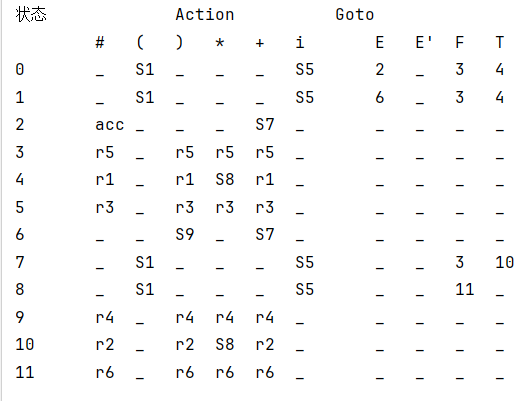
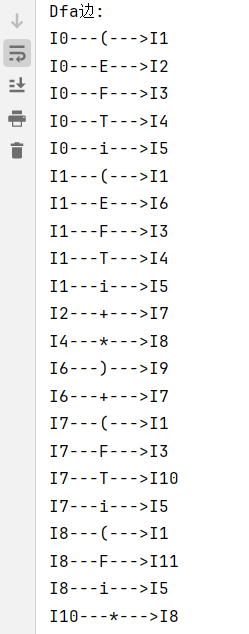
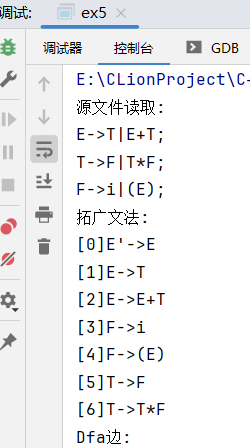
输出：1,2,+ 3

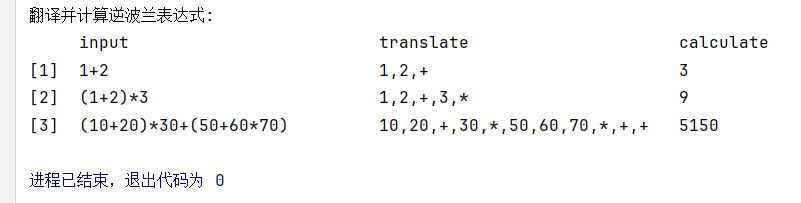
（2）(1+2)\*3;

输出：1,2,+,3,\* 9

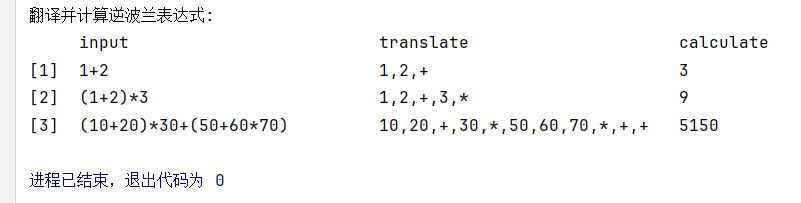
（3）(10+20)\*30+(50+60\*70)

输出：10,20,+30,\*50,60,70,\*,+,+ 5150





## 六、实验数据及处理结果



实验结果与预期一致

## 七、思考讨论题或体会或对改进实验的建议

### (1) 思考题

1. **语法制导翻译的工作方式？**

①词法分析：对输入的原始文本进行词法分析，将连续字符序列转换为Token流。

②语法分析：通过设计的语法规则，将词法分析得到的Token流转换成语法树。

③语义分析：对语法树进行语义分析，确保翻译的正确性和合理性。

④属性文法与目标代码生成：基于语法树和语义计算属性并生成目标语言的代码。

1. **为什么编译程序要设计生成中间代码方式？**

①提高可移植性：中间代码独立于目标机器的结构，使得编译器可以更容易地适应不同的CPU架构。这意味着同一个编译器前端可以为多个后端生成代码，从而提高了编译器的可移植性和灵活性。

②优化处理：中间代码为编译器提供了进行各种优化的机会，包括常量传播、无用代码删除、循环优化等。由于中间代码的形式通常比源代码更简单，因此更容易应用这些优化技术。

③简化编译过程：中间代码将复杂的编译过程分解为更小、更易管理的阶段，使得编译器的设计和实现更加模块化。这种分阶段的编译过程有助于提高编译器的可靠性和可维护性。

④支持多平台编译：由于中间代码与特定机器无关，因此可以在一个平台上生成中间代码，然后在另一个平台上进一步编译成目标代码。这在跨平台开发中非常有用。

⑤促进代码重用：中间代码可以作为不同编程语言之间的桥梁，使得一种语言编写的代码可以更容易地转换为另一种语言的代码。这对于代码重用和互操作性非常重要。

⑥便于调试和分析：中间代码通常具有更简单的结构，这使得编译器开发者可以更容易地分析和调试编译器的行为。

⑦提高执行效率：虽然直接生成机器代码可能看起来更快，但中间代码允许编译器进行更深入的优化，从而提高最终生成的目标代码的执行效率。

### (2) 实验体会

1. **对语法制导翻译的理解**：
   * 在实验过程中，我进一步理解了词法分析、语法分析和语义分析的相互关系。每个步骤都为下一步提供了基础，最终实现从源代码到机器可执行代码的转换。
   * 通过使用拓广文法、项目集规范簇DFA以及构建Action-Goto表的方法，我学会了如何在LR(1)的基础上优化并简化翻译过程。这种结构化的方法确保了翻译的准确性和效率。
2. **实践与理论的结合**：
   * 在编写程序的过程中，我应用了之前在课堂上学到的各种编译原理知识。例如，如何设计递归下降子程序来处理不同类型的文法产生式。
   * 通过实际编程，我验证了不同数据结构如栈、树、图等在处理语法树和属性文法时的有效性。这些工具帮助我实现了从输入串到逆波兰表达式的自动生成。
3. **调试技巧的提升**：
   * 在测试程序时，我遇到了一些错误，例如表达式无法正确匹配或生成的中间代码存在漏洞。通过反复调试，我提高了对程序行为的理解和问题解决能力。
   * 利用调试信息和错误提示，我逐步定位并修复了问题，从而增强了对程序运行机制的掌握。
4. **未来改进方向**：
   * 尽管实验取得了成功，但我发现仍有提升空间，比如进一步优化程序性能和用户体验。
   * 我计划在未来的工作中继续研究如何更高效地处理复杂表达式，并探索新的翻译技术和优化策略。

总的来说，通过这次实验，我不仅巩固了编译原理的理论知识，还提升了实际操作技能。这为我将来从事相关工作打下了坚实的基础。

## 八、参考资料

《编译原理课程实践任务书》

**实验六** **语法制导的三地址代码生成**

学生姓名：马星 学 号：5418122020 专业班级： 计算机科学与技术(卓越工程师)221班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期：2024/12/20 实验成绩：

## 一、实验项目名称

语法制导的三地址代码生成

## 二、实验目的

掌握计算机语言的语法分析程序设计与属性文法应用的实现方法。

编制一个能够进行语法分析并生成三地址代码的微型编译程序。

## 三、实验基本原理

1、考虑下述语法制导定义中文法，采用递归子程序法，改写文法，构造语法分析程序；

2、考虑下述语法制导定义中语义规则，改写语法分析程序，构造三地址代码生成程序。

|  |  |
| --- | --- |
| 产生式 | 语义规则 |
| S->id = E ; | S.code = E.code || gen(id.place’:=’E.place) |
| S->if C then S | C.true = newlabel;  C.false = S.next;  S1.next = S.next;  S.code = C.code || gen(E.true’:’) || S1.code |
| S->while C do S | S.begin = newlabel;  C.true = newlabel;  C.false = S.next;  S1.next = S.begin;  S.code = gen(S.begin’:’) || C.code ||  gen(E.true’:’) || S1.code || gen(‘goto’S.begin); |
| C->E1 > E2 | C.code = E1.code || E2.code ||  gen(‘if’E1.place’>’E2.place’goto’C.true) ||  gen(‘goto’C.false) |
| C->E1 < E2 | C.code = E1.code || E2.code ||  gen(‘if’E1.place’<’E2.place’goto’C.true) ||  gen(‘goto’C.false) |
| C->E1 = E2 | C.code = E1.code || E2.code ||  gen(‘if’E1.place’=’E2.place’goto’C.true) ||  gen(‘goto’C.false) |
| E->E1 + T | E.place = newtemp;  E.code = E1.code||T.code||  gen(E.place’:=’E1.place’+’T.place) |
| E->E1 - T | E.place = newtemp;  E.code = E1.code || T.code ||  gen(E.place’:=’E1.place’-’T.place) |
| T->T1 \* F | T.place = newtemp;  T.code = T1.code || F.code ||  gen(T.place’:=’T1.place’\*’F.place) |
| T->T1 / F | T.place = newtemp;  T.code = T1.code || F.code ||  gen(T.place’:=’T1.place’/’F.place) |
| F->( E ) | F.place = E.place;  F.code = E.code |
| F->id | F.place = id.name;  F.code = ‘ ‘ |
| F->int8 | F.place = int8.value;  F.code = ‘ ‘ |
| F->int10 | F.place = int10.value;  F.code = ‘ ‘ |
| F->int16 | F.place = int16.value;  F.code = ‘ ‘ |

## 四、主要仪器设备及耗材

PC微机

DOS操作系统或 Windows 操作系统

Turbo C 程序集成环境或 Visual C++ 程序集成环境

## 五、实验步骤（完整内容见光盘）

### 1、考虑给定的文法，消除左递归，提取左因子。

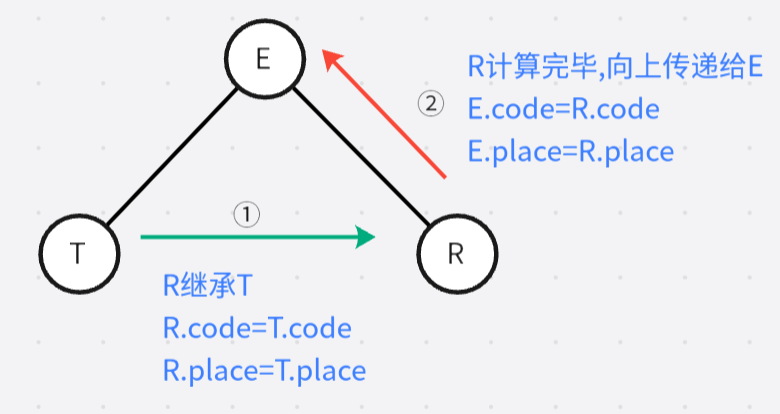
(1) E->E+T|E-T 变为 E->TR、R->+TR|-TR|ε

(2) T->T\*F|T/F 变为 T->FD、D->\*FD|/FD|ε

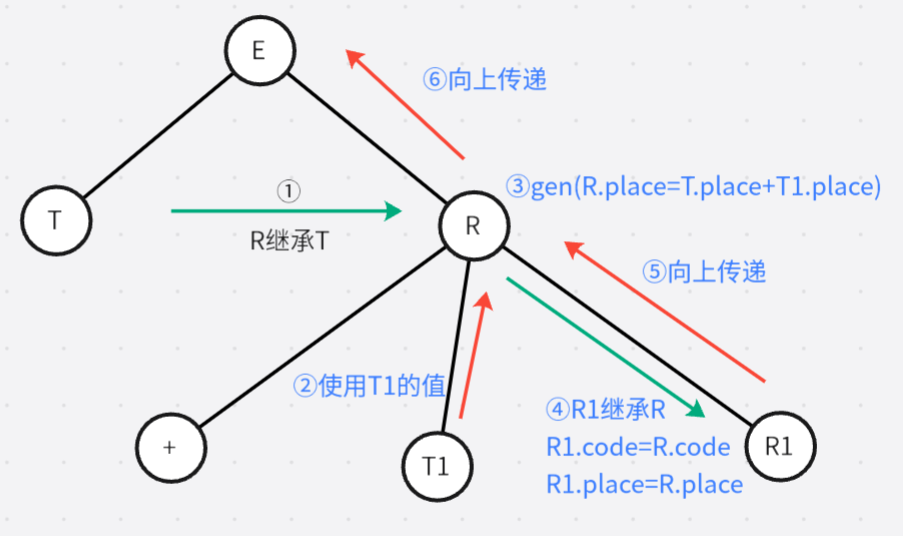
### 2、编制并化简语法图

(1) E->E+T|E-T 变为 E->TR、R->+TR|-TR|ε

对于E->TR,可以将T的属性作为继承属性传递给R, R计算完成后再向上传递给E



对于R->+ T1 R1, 在T1完成计算后, T+T1可以生成三地址码, 再将R的值传递给R1进行递归, R1递归完成后将值向上传递给R



对于R->ε, 此时为递归出口, 不需要做任何事, 因为继承和向上传递都在Procedure\_R中进行

(2) T->T\*F|T/F 变为 T->FD、D->\*FD|/FD|ε

与(1)结构是一致的, 做法相同

(3)最终的语义规则表

标注为红色部分的是修改的产生式和语义规则

|  |  |
| --- | --- |
| 产生式 | 语义规则 |
| S->id = E ; | S.code = E.code || gen(id.place’:=’E.place) |
| S->if C then S1 | C.true = newlabel;  C.false = S.next;  S1.next = S.next;  S.code = C.code || gen(E.true’:’) || S1.code |
| S->while C do S1 | S.begin = newlabel;  C.true = newlabel;  C.false = S.next;  S1.next = S.begin;  S.code = gen(S.begin’:’) || C.code ||  gen(E.true’:’)||S1.code || gen(‘goto’S.begin); |
| C->E1 > E2 | C.code = E1.code || E2.code ||  gen(‘if’E1.place’>’E2.place’goto’C.true) ||  gen(‘goto’C.false) |
| C->E1 < E2 | C.code = E1.code || E2.code ||  gen(‘if’E1.place’<’E2.place’goto’C.true) ||  gen(‘goto’C.false) |
| C->E1 = E2 | C.code = E1.code || E2.code ||  gen(‘if’E1.place’=’E2.place’goto’C.true) ||  gen(‘goto’C.false) |
| E->TR | R.code=T.code;  R.place=T.place;  E.place=R.place;  E.code=R.code; |
| R -> + T1 R1 | R.place=newtemp;  R.code=T.code||T1.code|| gen(R.place’:=’T.place’+’T1.place);  R1.code=R.code;  R1.place=R.place;  R.code=R1.code;  R.place=R1.place; |
| R -> - T1 R1 | R.place=newtemp;  R.code=T.code||T1.code||  gen(R.place’:=’T.place’-’T1.place);  R1.code=R.code;  R1.place=R.place;  R.code=R1.code;  R.place=R1.place; |
| R->ε |  |
| T-> F D | D.place=F.place;  D.code=F.code;  T.place=D.place;  T.code=D.code; |
| D->\* F D1 | D.place = newtemp;  D.code = F.code||F1.code||  gen(D.place’:= ’F.place’\*’F1.place);  D1.place=D.place;  D1.code=D.code;  D.place=D1.place;  D.code=D1.code; |
| D->/ F D1 | D.place = newtemp;  D.code = F.code||F1.code||  gen(D.place’:= ’F.place’/’F1.place);  D1.place=D.place;  D1.code=D.code;  D.place=D1.place;  D.code=D1.code; |
| D->ε |  |
| F->( E ) | F.place = E.place;  F.code = E.code |
| F->id | F.place = id.name;  F.code = ‘ ‘ |
| F->int8 | F.place = int8.value;  F.code = ‘ ‘ |
| F->int10 | F.place = int10.value;  F.code = ‘ ‘ |
| F->int16 | F.place = int16.value;  F.code = ‘ ‘ |

### 3、编制各个递归子程序函数

为每个非终结符定义递归程序

void Pro\_S() {

if (matchToken(Identifier)) {// S -> id=E;

advance();

if (!matchToken(Symbol, "=")) error();

advance();

Pro\_E();

if (!matchToken(Symbol, ";")) error();

advance();

} else if (matchToken(Keyword, "if")) { // S -> if C then S

advance();

Pro\_C();

if (!matchToken(Keyword, "then")) error();

advance();

Pro\_S();

} else if (matchToken(Keyword, "while")) { //S -> while C do S1

advance();

Pro\_C();

if (!matchToken(Keyword, "do"))error();

advance();

Pro\_S();

} else {

error();

}

}

void Pro\_C() {// C -> E1 ? E2

Pro\_E();

if (matchToken(Symbol, ">")) { // C -> E1 > E2

advance();

Pro\_E();

} else if (matchToken(Symbol, "<")) { // C -> E1 < E2

advance();

Pro\_E();

} else if (matchToken(Symbol, "=")) { // C -> E1 = E2

advance();

Pro\_E();

} else {

error();

}

}

void Pro\_E() { // E -> TR

Pro\_T();

Pro\_R();

}

void Pro\_R() { // R -> ? T R

if (matchToken(Symbol, "+")) {//R -> + T1 R1

advance();

Pro\_T();

Pro\_R();

} else if (matchToken(Symbol, "-")) {//R -> - T1 R1

advance();

Pro\_T();

Pro\_R();

} else { // R -> ε

}

}

void Pro\_T() {// T -> F D

Pro\_F();

Pro\_D();

}

void Pro\_D() {

if (matchToken(Symbol, "\*")) {//D -> \* F1 D1

advance();

Pro\_F();

Pro\_D();

} else if (matchToken(Symbol, "/")) {// D -> / F1 D1

advance();

Pro\_F();

Pro\_D();

} else { // D -> ε

}

}

void Pro\_F() {

if (matchToken(Symbol, "(")) { // F -> (E)

advance();

Pro\_E();

if (!matchToken(Symbol, ")")) error();

advance();

} else if (matchToken(Identifier)) { // F -> id

advance();

} else if (matchToken(DecInt) || matchToken(OctInt)

|| matchToken(HexInt)) {// F -> int

advance();

} else {

error();

}

}

void error() const {

cout << "无法匹配tokens[" << tokenIdx << "]: " << sym.value << "\n";

exit(-1);

}

### 4、连接实验一的词法分析函数scan( )，进行测试

先读入文件中的程序字符串, 然后使用实验二的词法分析器转换为token流,调用递归下降程序进行匹配

若匹配失败则报错并程序退出, 匹配成功则打印”成功识别”

int main() {

string sourceCode = read("../Ex6/g6.txt");// 读文件

WordFactor wordFactor;

vector<Token> tokens = wordFactor.scan(sourceCode);// 转换为token流

cout << "Tokens: ";

for (const auto &tok: tokens) {

cout << tok.value << " ";

}

cout << endl;

Recursive\_ recursive;

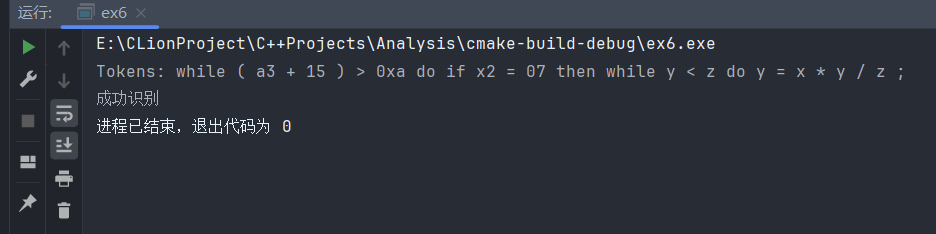
if (recursive.check(tokens)) { // 递归下降匹配

cout << "成功识别";

}

return 0;

}



### 5、设计三地址代码生成的数据结构和算法

#### (1) 定义树结构, 在进行递归下降时, 建立语法树

class TreeNode {

public:

string value; // 节点名

vector<TreeNode> children;// 子树

TreeNode() = default;

explicit TreeNode(const string &value) {

this->value = value;

}

void putChild(const TreeNode &node) {

children.push\_back(node);

}

void putChild(const string &val) {

children.emplace\_back(val);

}

};

#### (2) 为每个非终结符定义一个对应的类

类内带有属性, 在进行递归下降时进行属性的计算

class S {

public :

string begin, // while的起始位置

next, // 指示的下一段程序

code; // 三地址码

};

class C {

public:

string \_true,// 判断为true时的程序

\_false,// 判断为false时的程序

code;// 三地址码

};

typedef class UnName {

public:

string code,//三地址码

place;// 入口地址

} E, T, R, F, D;

### 6、将各个递归子程序函数改写为代码生成函数

在递归子程序中, 创建语法树进行返回

每个子程序将非终结符对应的对象传入, 插入继承属性和综合属性的计算

最后匹配完成时S.code就是生成的三地址代码

TreeNode Pro\_S(S &s) {

TreeNode tree("S");

if (matchToken(Identifier)) {// S -> id=E;

// id

string id = sym.value;

advance();

tree.putChild(id);

// =

if (!matchToken(Symbol, "=")) error();

advance();

tree.putChild("=");

// E

E e;

const TreeNode &tree\_e = Pro\_E(e);

tree.putChild(tree\_e);

// ;

if (!matchToken(Symbol, ";")) error();

advance();

tree.putChild(";");

// 计算code

s.code = e.code + "\t" + gen(id + " := " + e.place);

} else if (matchToken(Keyword, "if")) { // S -> if C then S

// if

advance();

tree.putChild("if");

// C

C c;

c.\_true = newlabel();

c.\_false = s.next;

const TreeNode &tree\_c = Pro\_C(c);

tree.putChild(tree\_c);

// then

if (!matchToken(Keyword, "then")) error();

advance();

tree.putChild("then");

// S

S s1;

s1.next = s.next;

s1.begin = c.\_true;

const TreeNode &tree\_s1 = Pro\_S(s1);

tree.putChild(tree\_s1);

// 计算code

s.code = c.code + gen(c.\_true + ":") + s1.code;

} else if (matchToken(Keyword, "while")) { //S -> while C do S1

bool beginEmpty = s.begin.empty();

if (beginEmpty) s.begin = newlabel();

// while

advance();

tree.putChild("while");

// C

C c;

c.\_true = newlabel();

c.\_false = s.next;

const TreeNode &tree\_c = Pro\_C(c);

tree.putChild(tree\_c);

// do

if (!matchToken(Keyword, "do"))error();

advance();

tree.putChild("do");

// S

S s1;

s1.next = s.begin;

const TreeNode &tree\_s1 = Pro\_S(s1);

tree.putChild(tree\_s1);

if (beginEmpty) s.code = gen(s.begin + ":");

s.code += c.code + gen(c.\_true + ":") + s1.code

+ "\t" + gen(genKeyword("goto") + s.begin);

}

return tree;

}

TreeNode Pro\_C(C &c) {// C -> E1 ? E2

TreeNode tree("C");

// E1

E e1;

const TreeNode &tree\_e1 = Pro\_E(e1);

tree.putChild(tree\_e1);

if (matchToken(Symbol, ">")) { // C -> E1 > E2

// >

advance();

tree.putChild(">");

// E2

E e2;

const TreeNode &tree\_e2 = Pro\_E(e2);

tree.putChild(tree\_e2);

// 计算code

c.code = e1.code + e2.code

+ "\t" + gen(genKeyword("if") + e1.place + " > " + e2.place

+ genKeyword("goto") + c.\_true)

+ "\t" + gen(genKeyword("goto") + c.\_false);

} else if (matchToken(Symbol, "<")) { // C -> E1 < E2

// <

advance();

tree.putChild("<");

// E2

E e2;

const TreeNode &tree\_e2 = Pro\_E(e2);

tree.putChild(tree\_e2);

// 计算code

c.code = e1.code + e2.code

+ "\t" + gen(genKeyword("if") + e1.place + " < " + e2.place

+ genKeyword("goto") + c.\_true)

+ "\t" + gen(genKeyword("goto") + c.\_false);

} else if (matchToken(Symbol, "=")) { // C -> E1 = E2

// =

advance();

tree.putChild("=");

// E2

E e2;

const TreeNode &tree\_e2 = Pro\_E(e2);

tree.putChild(tree\_e2);

// 计算code

c.code = e1.code + e2.code

+ "\t" + gen(genKeyword("if") + e1.place + " = " + e2.place

+ genKeyword("goto") + c.\_true)

+ "\t" + gen(genKeyword("goto") + c.\_false);

} else {

error();

}

return tree;

}

TreeNode Pro\_E(E &e) { // E -> TR

TreeNode tree("E");

// T

T t;

const TreeNode &tree\_t = Pro\_T(t);

tree.putChild(tree\_t);

// R

R r;

r.code = t.code;// R继承T

r.place = t.place;

const TreeNode &tree\_r = Pro\_R(r);

tree.putChild(tree\_r);

// R计算完毕, 向上传递

e.code = r.code;

e.place = r.place;

return tree;

}

TreeNode Pro\_R(R &r) { // R -> ? T R

TreeNode tree("R");

if (matchToken(Symbol, "+")) {//R -> + T1 R1

// +

advance();

tree.putChild("+");

// T

T t1;

const TreeNode &tree\_t = Pro\_T(t1);

tree.putChild(tree\_t);

// 生成T + T1的三地址码

string t\_place = r.place, t\_code = r.code;

r.place = newtemp();

r.code = t\_code + t1.code + "\t" + gen(r.place + " := " + t\_place + " + " + t1.place);

// R1

R r1 = r; // 将R的属性传递给R1, 等待R1计算完后向上传递

const TreeNode &tree\_r1 = Pro\_R(r1);

tree.putChild(tree\_r1);

// R1计算完毕, 向上传递

r.code = r1.code;

r.place = r1.place;

} else if (matchToken(Symbol, "-")) {//R -> - T1 R1

// -

advance();

tree.putChild("-");

// T

T t1;

const TreeNode &tree\_t1 = Pro\_T(t1);

tree.putChild(tree\_t1);

string t\_place = r.place, t\_code = r.code;

r.place = newtemp();

r.code = t\_code + t1.code + "\t" + gen(r.place + " := " + t\_place + " - " + t1.place);

// R1

R r1 = r;

const TreeNode &tree\_r1 = Pro\_R(r1);

tree.putChild(tree\_r1);

// R1计算完毕, 向上传递

r.code = r1.code;

r.place = r1.place;

} else { // R -> ε

tree.putChild(EMPTY);

}

return tree;

}

TreeNode Pro\_T(T &t) {// T -> F D

TreeNode tree("T");

// F

F f;

const TreeNode &tree\_f = Pro\_F(f);

tree.putChild(tree\_f);

// D

D d;

d.code = f.code;// D继承F

d.place = f.place;

const TreeNode &tree\_d = Pro\_D(d);

tree.putChild(tree\_d);

// D计算完毕, 向上传递

t.code = d.code;

t.place = d.place;

return tree;

}

TreeNode Pro\_D(D &d) { // D -> ? F D

TreeNode tree("D");

if (matchToken(Symbol, "\*")) {//D -> \* F1 D1

// \*

advance();

tree.putChild("\*");

// F

F f1;

const TreeNode &tree\_f = Pro\_F(f1);

tree.putChild(tree\_f);

string f\_place = d.place, f\_code = d.code;

d.place = newtemp();

d.code = f\_code + f1.code + "\t" + gen(d.place + " := " + f\_place + " \* " + f1.place);

// D1

D d1 = d; //D1继承D, 等待D1计算完毕后向上传递

const TreeNode &tree\_d = Pro\_D(d1);

tree.putChild(tree\_d);

// D1计算完毕, 向上传递

d.code = d1.code;

d.place = d1.place;

} else if (matchToken(Symbol, "/")) {// D -> / F1 D1

// /

advance();

tree.putChild("/");

// F

F f1;

const TreeNode &tree\_f = Pro\_F(f1);

tree.putChild(tree\_f);

string f\_place = d.place, f\_code = d.code;

d.place = newtemp();

d.code = f\_code + f1.code + "\t" + gen(d.place + " := " + f\_place + " / " + f1.place);

// D1

D d1 = d; //D1继承D, 等待D1计算完毕后向上传递

const TreeNode &tree\_d1 = Pro\_D(d1);

tree.putChild(tree\_d1);

// D1计算完毕, 向上传递

d.code = d1.code;

d.place = d1.place;

} else { // D -> ε

tree.putChild("ε");

}

return tree;

}

TreeNode Pro\_F(F &f) { // F -> (E)|id|int

TreeNode tree("F");

if (matchToken(Symbol, "(")) { // F -> (E)

// (

advance();

tree.putChild("(");

// E

E e;

const TreeNode &tree\_e = Pro\_E(e);

tree.putChild(tree\_e);

// )

if (!matchToken(Symbol, ")")) error();

advance();

tree.putChild(")");

f.place = e.place;

f.code = e.code;

} else if (matchToken(Identifier)) {// F -> id

// id

string id = sym.value;

advance();

TreeNode tree\_id("id");

tree\_id.putChild(id);

tree.putChild(tree\_id);

f.place = id;

f.code = "";

} else if (matchToken(DecInt) || matchToken(OctInt)

|| matchToken(HexInt)) {// F -> int

// int

string data = sym.value;

advance();

TreeNode tree\_int("int");

tree\_int.putChild(data);

tree.putChild(tree\_int);

f.place = data;

f.code = "";

} else {

error();

}

return tree;

}

void error() const {

cout << "无法匹配tokens[" << ptr << "]: " << sym.value << "\n";

exit(-1);

}

### 7、调试程序：输入一个语句，检查输出的三地址代码

读入程序, 转换为token流, 输入递归下降程序中进行转换, 输出转换后的三地址代码

int main() {

string sourceCode = read("../Ex6/g6.txt");

WordFactor wordFactor;

vector<Token> tokens = wordFactor.scan(sourceCode);

cout << "Tokens: ";

for (const auto &tok: tokens) {

cout << tok.value << " ";

}

cout << endl;

Recursive recursive;

string res = recursive.convert(tokens);

cout << "三地址码:" << endl;

cout << res;

return 0;

}

/\*\*

\* 解析Token流, 转换为三地址码

\* @param tokens Token流表示的一段程序

\* @return 三地址码

\*/

string convert(vector<Token> &tokens) {

this->tokens = tokens;

this->tokens.emplace\_back(Undefined, END);// 末尾加上终结符

S s;

s.next = newlabel();// L0: S.next

advance(); // 读入第一个token

Pro\_S(s); // 递归下降进行解析

// 解析完成

if (sym.type != Undefined) error();// 还有未解析的token

return s.code + s.next + ":\t" + colorString("//S.next", GREY);

}

输入数据例：while (a3+15)>0xa do if x2 = 07 then while y<z do y = x \* y / z;

正确结果：等效的三地址代码序列

L1: t1 := a3 + 15

if t1 > 10 goto L2

goto L0

L2:

L3:

if x2 > 7 goto L4

goto L1

L4: if y < z goto L5

goto L1

L5: t2 = x \* y

t3 = t2 / z

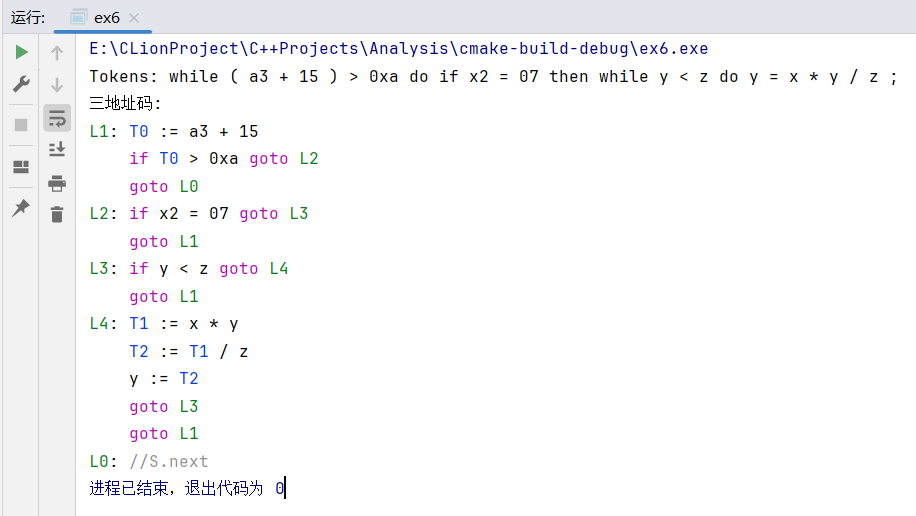
y = t3

goto L3

goto L1

L0: // S.next

我的运行输出:



## 六、实验数据及处理结果

答案结果：

L1: t1 := a3 + 15

if t1 > 10 goto L2

goto L0

L2:

L3:

if x2 > 7 goto L4

goto L1

L4: if y < z goto L5

goto L1

L5: t2 = x \* y

t3 = t2 / z

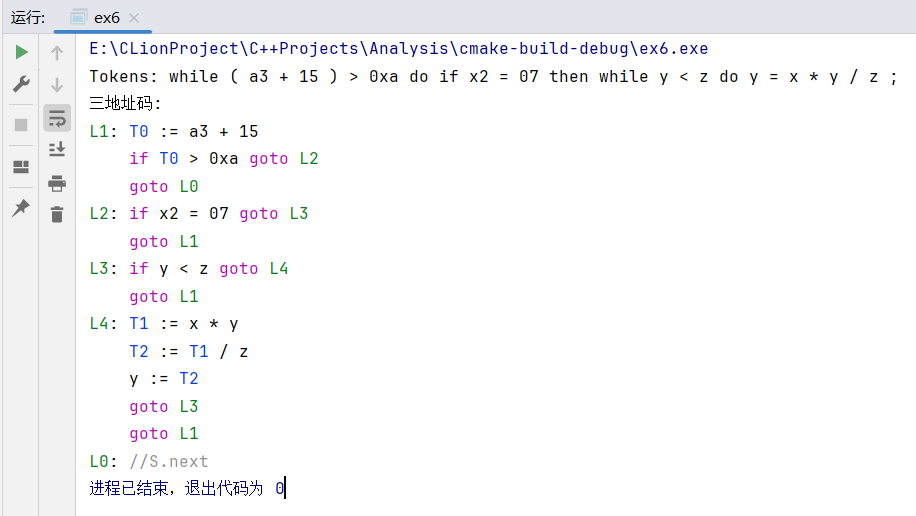
y = t3

goto L3

goto L1

L0: // S.next

我的输出:



实验结果与给出的答案有一定区别, 但不会影响正确性, 区别如下:

(1) 0xa,答案将其替换为了10进制, 我的程序没有做这件事, 我的程序如果要进行转换可以在F->int时对int的进制进行识别, 再做对应的进制转换

(2) 答案中的x2 > 7有误, 应该为x2 = 7

(3) 答案的L有6个,而我的程序只有5个, 这是因为答案的L2是一个空标签, 而我的程序略过了这个空标签

打印在控制台有不同颜色是一个终端的颜色格式

#define NONE "\e[0m"

#define BLACK "\e[30m"

#define RED "\e[31m"

#define GREEN "\e[32m"

#define YELLOW "\e[33m"

#define BLUE "\e[34m"

#define PURPLE "\e[35m"

#define BABYBLUE "\e[36m"

#define GREY "\e[37m"

/\*\*

\* 给字符串添加颜色, 打印时更好看

\* @param s 字符串

\* @param color 颜色(见上表define)

\*/

static string colorString(const string &s, const string &color) {

return color + s + NONE;

}

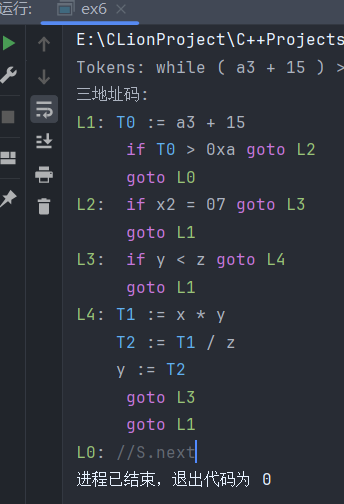
## 七、思考讨论题或体会或对改进实验的建议

### (1) 思考题

#### 1、生成的三地址代码可否直接输出（不采用数据结构来实现属性code）?

可以, 因为属性code是按序拼接而成的, 所以只要输出顺序正确即可不使用数据结构进行存储

如图, 我将我的程序中的code属性全部删除, 替换为cout输出, 再略微调整顺序, 输出结果与之前一致



#### 2、如何保证四则运算的优先关系和左结合性？

  优先级：通过改造文法，引入中间变量T来确保乘除的递归深度更大，从而先递归掉乘法和除法再递归加法和减法，保证优先级正确

 左结合性：通过递归顺序,例如R->+ T1 R1, 在解析完T1后先生成R节点对应的三地址码再继续递归R1，保证先生成R的三地址码，这个时候便可满足左结合性

#### 3、如何采用代码段相对地址代替三地址代码序列中的标号？

可以重新扫描一遍生成的三地址码，确定每一个标号对应的相对地址进行替换

### (2) 实验体会

1. **理解和应用文法转换技术**：
   * 通过学习LL(1)文法的构造方法，我掌握了将非LL(1)文法转换为LL(1)文法的技巧，包括消除左递归和使用提取左因子的方法来减少回溯。这些技术不仅在实验中得到了实际应用，还使我对文法的结构和性质有了更深入的理解。
2. **设计和实现递归下降分析程序：**
   * 在编写递归下降分析程序时，我学会了如何根据文法规则设计递归函数，以处理不同类型的语法结构。这个过程让我更加熟悉了递归思想和自顶向下的分析方法。同时，通过调试和优化程序，我提高了编程技巧和问题解决能力。
3. **生成三地址代码：**
   * 在将文法转换为三地址代码的过程中，我深刻体会到中间表示形式的重要性。三地址代码作为一种中间表示，既简化了后续的优化过程，又为最终的目标代码生成提供了便利。通过实际操作，我学会了如何在每个非终结符的处理过程中插入适当的语义动作，生成正确的三地址代码。
4. **综合应用与调试：**
   * 整个实验过程中，我不断进行调试和测试，确保每个步骤的正确性。从输入源程序到输出三地址代码，每一步都经过了严格的验证。这不仅提高了我对编译器设计的理解，也增强了我的耐心和细心。

总的来说，这次实验让我对编译原理课程中的理论知识有了更深刻的理解，同时也提升了我在程序设计、调试和优化方面的技能。我相信这些经验和技能将为我未来的学习和工作提供宝贵的支持。

## 八、参考资料

《编译原理课程实践任务书》