**编译原理**

班级：21计算机科学与技术留学生（1）

姓名：LOW REN HONG 学号：2021529620004

实验二 自顶向下语法分析器的构建

1. 实验要求

运用LL(1)分析法，针对给定的上下文无关文法，给出实验方案。预估实验中可能出现的问题。

1. 实验方案

（评价依据实验方案设计是否合理，包括输入输出的设计）

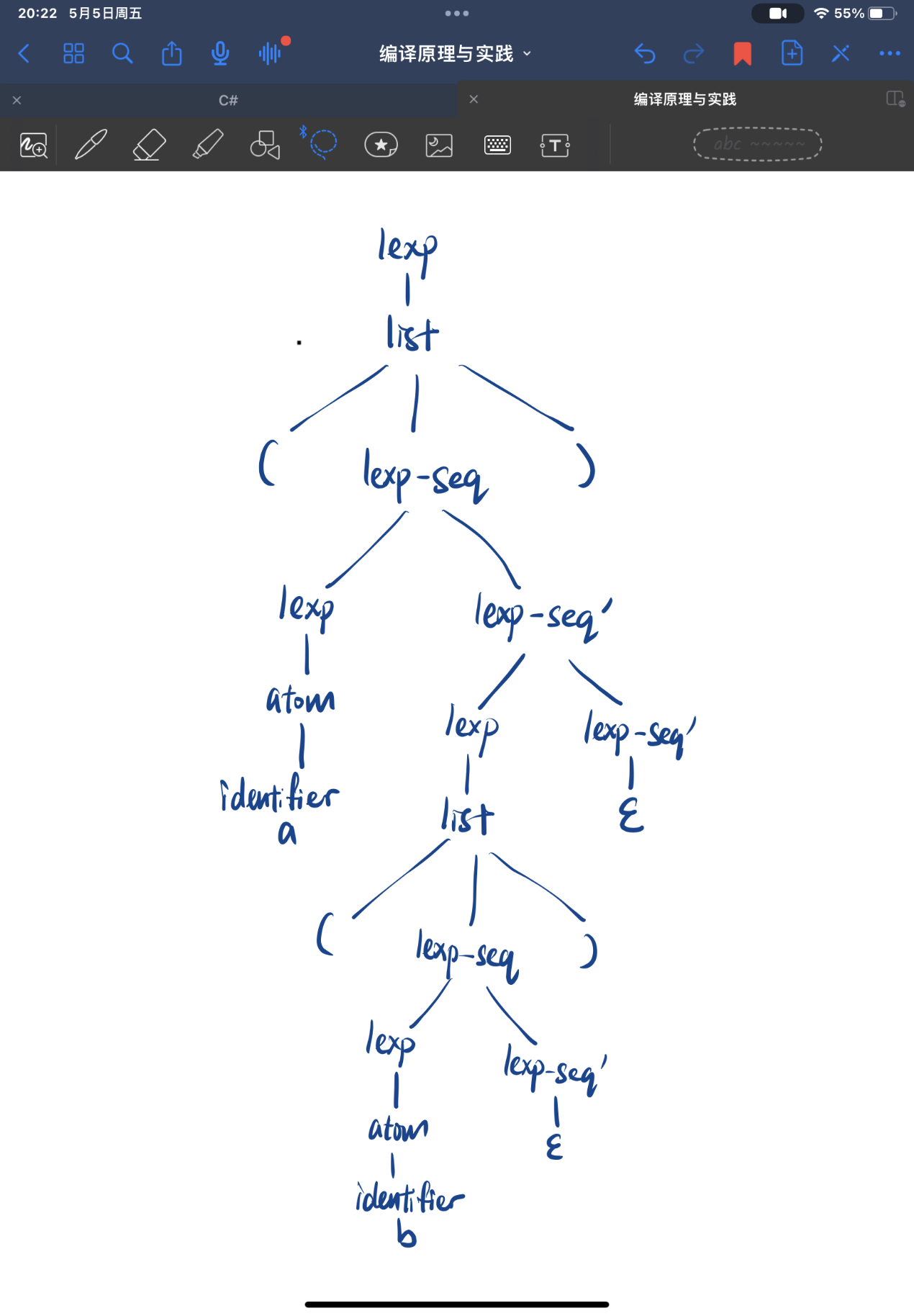
1. 构造LL(1)分析表

分析给定文法，消除左递归及提取左因子，以使文法符合LL(1)文法。计算First集合、Follow集合，并根据First集合和Follow集合构造LL(1)分析表。

1. 输入输出设计

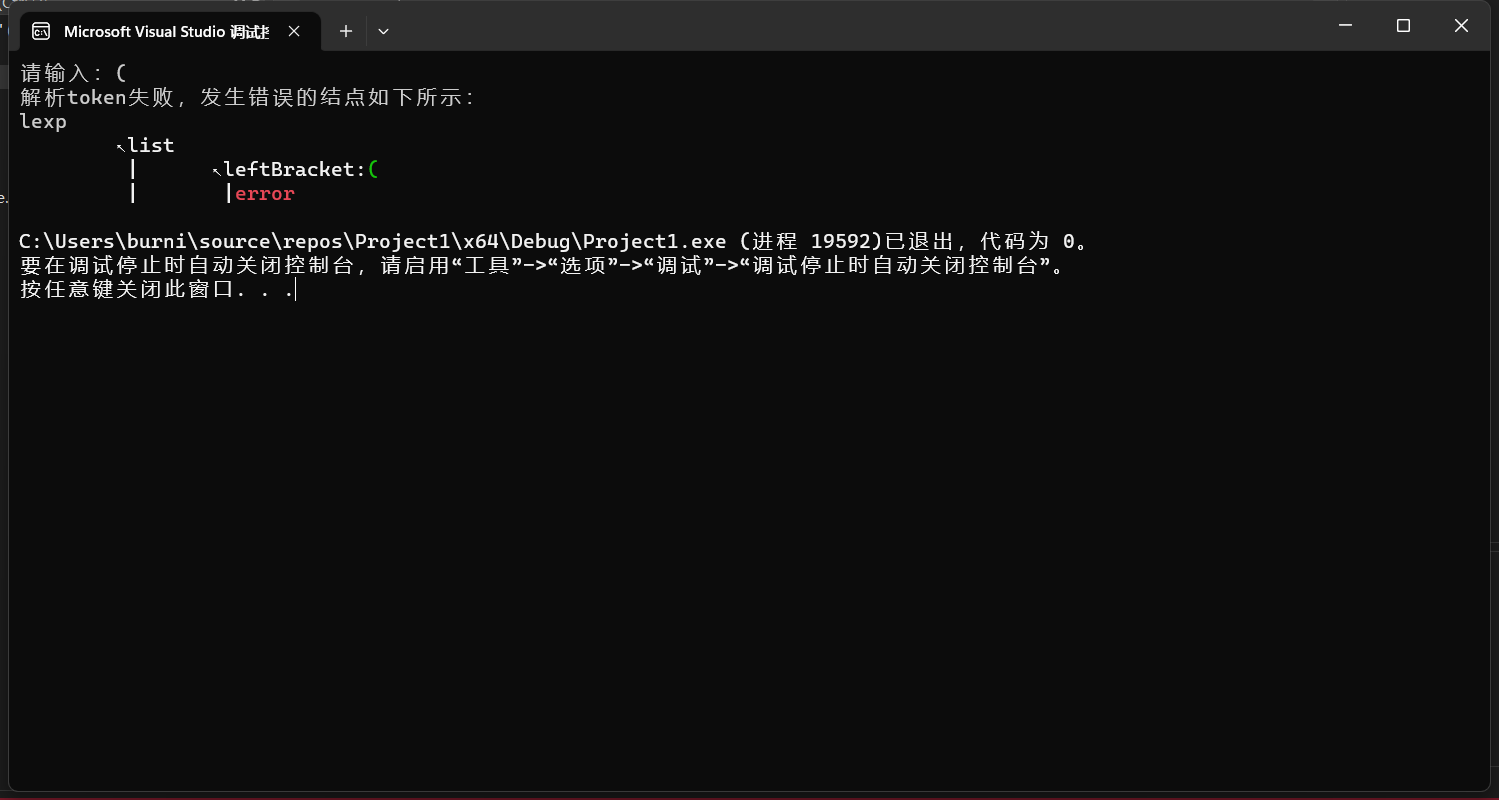
严格来说，输入的字符串首先通过词法分析得到Token序列，再经过本算法得到分析树。但是本次实验的输入较为简单，且侧重点在于后者，故仅对输入串进行简单的判断，生成Token序列。输入字符串中的单个字符表示一个Token。一个字母即是一个identifier，一个数字即为一个number。

  输出形式为解析得到的语法分析树。以缩进形式表示树的父子关系。例如，对于下左图所示的分析树，输出形式为下右图所示。



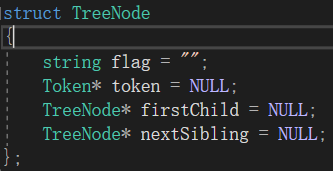


若解析失败，输出失败提示，并将分析树输出，以显示发生错误的结点位置，如下图

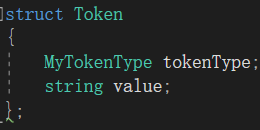


1. 数据结构设计

程序使用树来存储得到的分析树，由于该分析树是一般的树状结构，采取树的孩子兄弟表示法。从而用二叉树的存储结构来表示树的逻辑结构。结点设计如下



其中Token为算法输入的数据结构，设计如下



1. 预估问题

(1)预估问题（是否有预估的问题，预估的问题是否合理）

给定文法可能不是LL(1)文法，或无法构造出LL(1)分析表。

(2) 理论基础（评价依据 理论知识非常清楚）

* First集合的定义

令X为一个文法符号或ε，则集合First(X)的定义如下：

1.若X是终结符或ε，则First(X) = {X}。

2.若X是非终结符，则对于每个产生式 X→X1X2……Xn，First(X)都包含了First(X1) - {ε}。若对于某个i<n，所有的集合First(X1), …, First(Xi)都包括了ε, 则First(X)也包括了First(Xi+1) - {ε}。若所有集合First(X1), …, First(Xn)包括了ε，则First(X)也包括ε。

* Follow集合的定义

集合Follow(A)的定义如下：

1.若A是开始符号，则$就在Follow(A)中。

2.若存在产生式B->αAγ，则First(γ) - {ε}在Follow(A)中。

3.若存在产生式B->αAγ，且ε在First(γ)中，则Follow(A)包括Follow(B)。

* LL(1)文法的定义

定义：如果文法G相关的LL(1)分析表的每个项目中至多只有一个产生式，则该文法就LL(1)文法。

定理：若满足以下条件，则BNF中的文法就是LL(1)文法：

1.在每个产生式A->α1|α2|…|αn中，对于所有的i和j：1<=i，j<=n，i!=j，First(αi)∩First(αj)为空。

2.对于每个非终结符A,若First(A)包含了ε，那么First(A)nFollow(A)为空。

* LL(1)分析表的构造方法

LL(1)分析表M[N, T]的构造：为每个非终结符A和产生式A->α重复以下两个步骤：

1.对于First(α)中的每个记号a，都将->α添加到项目M[A, a]中。

2.若ε在First(α)中，则对于Follow(A)的每个元素a（记号或是$），都将A->α添加到M[A, a]中。

1. 内容和步骤
2. 针对4.8习题输入和输出的设计及代码

* 习题4.8文法

Lexp->atom|list

atom ->number|identifier

list->(lexp-seq)

lexp-seq->lexp-seq lexp|lexp

* 消除左递归

Lexp->atom|list

atom ->number|identifier

list-> (lexp-seq)

lexp-seq->lexp lexp-seq’

lexp-seq’ ->lexp lexp-seq’|ε

* 计算First集合

First(lexp) = { number, identifier, ( }

First(atom) = { number, identifier }

First(list) = { ( }

First(lexp-seq) = { number, identifier, ( }

First(lexp-seq’) = { number, identifier, ( ,ε}

* 计算Follow集合

Follow(lexp) = { $, number, identifier, (, ) }

Follow(atom) = { $, number, identifier, (, ) }

Follow(list) = { $, number, identifier, (, ) }

Follow(lexp-seq) = { ) }

Follow(lexp-seq’) = { ) }

* 构造LL(1)分析表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| M[N,T] | number | identifier | ( | ) | $ |
| lexp | Lexp->atom | lexp->atom | lexp->list |  |  |
| atom | atom->number | atom->identifier |  |  |  |
| list |  |  | list-> (lexp-seq) |  |  |
| lexp-seq | lexp-seq->lexp lexp-seq’ | lexp-seq->lexp lexp-seq’ | lexp-seq->lexp lexp-seq’ |  |  |
| lexp-seq’ | lexp-seq’ ->lexp lexp-seq’ | lexp-seq’ ->lexp lexp-seq’ | lexp-seq’ ->lexp lexp-seq’ | lexp-seq’ ->ε |  |

2. 针对现场给定语法的设计和处理

* 给定文法：

E->E+T|T

E->T\*F|F

F-> (E)|id

* 提取左因子

E->TE’

E’ ->+TE’|ε

T->FT’

T’ ->\*FT’|ε

F-> (E)|id

* 计算First集合

First(E) = { (, id }

First(E’) = { +,ε}

First(T’) = { (, id }

First(T’) = { \*,ε}

First(F) = { (, id }

* 计算Follow集合

Follow(E) = { $, ) }

Follow(E’) = { $, ) }

Follow(T) = { +, $, ) }

Follow(T’) = { +, $, ) }

Follow(F) = { \*, +, $, ) }

* 构造LL(1)分析表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| M[N,T] | id | ( | ) | + | - | $ |
| E | E->TE’ | E->TE’ |  |  |  |  |
| E’ |  |  | E’ ->ε |  |  | E’ ->ε |
| T | T->FT’ | T->FT’ |  |  |  |  |
| T’ |  |  | T’ ->ε | T’ ->ε | T’ ->\*FT’ | T’ ->ε |
| F | F->id | F-> (E) |  |  |  |  |

2、实验具体步骤

1. 分析文法，消除左递归并提取左因子，使文法符合LL(1)文法
2. 计算First集合及Follow集合
3. 构造LL(1)分析表
4. 根据LL(1)分析表设计代码
5. 运行与调试
6. 实验结果:
7. 代码

* 针对习题4.8代码

#include <iostream>

#include <iomanip>

#include <string>

#include <windows.h>

using namespace std;

string typeName[6] = { "undefine","number","identifier","leftBracket","rightBracket","endInput" };

enum MyTokenType

{

undefine, number, identifier, leftBracket, rightBracket, endInput

};

struct Token

{

MyTokenType tokenType;

string value;

};

//采用孩子兄弟表示法的树节点

struct TreeNode

{

string flag = "";

Token\* token = NULL;

TreeNode\* firstChild = NULL;

TreeNode\* nextSibling = NULL;

};

void input();

TreeNode\* lexp();

TreeNode\* atom();

TreeNode\* list();

TreeNode\* lexp\_seq();

TreeNode\* lexp\_seq1();

TreeNode\* match();

TreeNode\* errorNode();

void printTree(TreeNode\* head, int depth = 0, bool isFirst = false);

Token inputToken[1024];

int tokenNumber = 0;

int curPos = 0;

bool error = false;

int main() {

input();

curPos = 0;

while (inputToken[curPos].tokenType != endInput && !error)

{

TreeNode\* head = lexp();

if (error)

cout << "解析token失败，发生错误的结点如下所示：" << endl;

printTree(head);

}

return 0;

}

//可将输入按实验一识别为Token字符串，但这里仅做简单判断(一个字符为一个Token)

void input() {

cout << "请输入：";

char ch;

while ((ch = getchar()) != '\n') {

inputToken[tokenNumber].value = ch;

if (ch == '(')

inputToken[tokenNumber].tokenType = leftBracket;

else if (ch == ')')

inputToken[tokenNumber].tokenType = rightBracket;

else if (ch >= '0' && ch <= '9')

inputToken[tokenNumber].tokenType = number;

else if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))

inputToken[tokenNumber].tokenType = identifier;

else

inputToken[tokenNumber].tokenType = undefine;

tokenNumber++;

}

inputToken[tokenNumber++].tokenType = endInput;

}

TreeNode\* lexp() {

TreeNode\* node = new TreeNode;

node->flag = "lexp";

switch (inputToken[curPos].tokenType)

{

case number:

case identifier:

node->firstChild = atom();

break;

case leftBracket:

node->firstChild = list();

break;

default:

node = errorNode();

error = true;

break;

}

return node;

}

TreeNode\* atom() {

TreeNode\* node = new TreeNode;

node->flag = "atom";

switch (inputToken[curPos].tokenType)

{

case number:

case identifier:

node->firstChild = match();

break;

default:

node = errorNode();

error = true;

break;

}

return node;

}

TreeNode\* list() {

TreeNode\* node = new TreeNode;

node->flag = "list";

switch (inputToken[curPos].tokenType)

{

case leftBracket: {

TreeNode\* tNode = node->firstChild = match();

tNode = tNode->nextSibling = lexp\_seq();

if (error)

return node;

tNode = tNode->nextSibling = match();

break;

}

default:

node = errorNode();

error = true;

break;

}

return node;

}

TreeNode\* lexp\_seq() {

TreeNode\* node = new TreeNode;

node->flag = "lexp\_seq";

switch (inputToken[curPos].tokenType)

{

case number:

case identifier:

case leftBracket: {

TreeNode\* tNode = node->firstChild = lexp();

if (error)

return node;

tNode = tNode->nextSibling = lexp\_seq1();

break;

}

default:

node = errorNode();

error = true;

break;

}

return node;

}

TreeNode\* lexp\_seq1() {

TreeNode\* node = new TreeNode;

node->flag = "lexp\_seq'";

switch (inputToken[curPos].tokenType)

{

case number:

case identifier:

case leftBracket: {

TreeNode\* tNode = node->firstChild = lexp();

if (error)

return node;

tNode = tNode->nextSibling = lexp\_seq1();

break;

}

case rightBracket:

(node->firstChild = new TreeNode)->flag = "empty";

break;

default:

node = errorNode();

error = true;

break;

}

return node;

}

TreeNode\* match() {

TreeNode\* newNode = new TreeNode;

newNode->flag = "match";

newNode->firstChild = newNode->nextSibling = NULL;

newNode->token = &(inputToken[curPos++]);

return newNode;

}

TreeNode\* errorNode() {

TreeNode\* errorNode = new TreeNode;

errorNode->flag = "error";

errorNode->firstChild = errorNode->nextSibling = NULL;

errorNode->token = &(inputToken[curPos++]);

return errorNode;

}

void printTree(TreeNode\* head, int depth /\*defalut=0\*/, bool isFirst /\*default=false\*/) {

if (head == NULL)

return;

int i;

for (i = 0; i < depth - 1; i++)

cout << "\t |";

if (i < depth)

cout << (isFirst ? "\t↖" : "\t |");

if (head->flag == "match") {

MyTokenType type = head->token->tokenType;

string value = head->token->value;

cout << typeName[type] << ':';

//匹配，改变颜色为绿色

SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_GREEN);

cout << value << endl;

//改回输出颜色

SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

}

else

{

//发生错误的结点，用红色输出

if (head->flag == "error")

SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED);

cout << head->flag << endl;

SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

printTree(head->firstChild, depth + 1, true);

}

printTree(head->nextSibling, depth);

}

* 针对习题4.8代码

#include <iostream>

#include <iomanip>

#include <string>

#include <windows.h>

using namespace std;

string typeName[7] = { "undefine","id","leftBracket","rightBracket","addop", "mulop","endInput" };

enum MyTokenType

{

undefine, id, leftBracket, rightBracket, addop, mulop, endInput

};

struct Token

{

MyTokenType tokenType;

string value;

};

//采用孩子兄弟表示法的树节点

struct TreeNode

{

string flag = "";

Token\* token = NULL;

TreeNode\* firstChild = NULL;

TreeNode\* nextSibling = NULL;

};

void input();

TreeNode\* E();

TreeNode\* E1();

TreeNode\* T();

TreeNode\* T1();

TreeNode\* F();

TreeNode\* match();

TreeNode\* errorNode();

void printTree(TreeNode\* head, int depth = 0, bool isFirst = false);

Token inputToken[1024];

int tokenNumber = 0;

int curPos = 0;

bool error = false;

int main() {

input();

curPos = 0;

while (inputToken[curPos].tokenType != endInput && !error)

{

TreeNode\* head = E();

if (error)

cout << "解析token失败，发生错误的结点如下所示：" << endl;

printTree(head);

}

return 0;

}

//可将输入按实验一识别为Token字符串，但这里仅做简单判断(一个字符为一个Token)

void input() {

cout << "请输入：";

char ch;

while ((ch = getchar()) != '\n') {

inputToken[tokenNumber].value = ch;

if (ch == '(')

inputToken[tokenNumber].tokenType = leftBracket;

else if (ch == ')')

inputToken[tokenNumber].tokenType = rightBracket;

else if (ch == '+')

inputToken[tokenNumber].tokenType = addop;

else if (ch == '\*')

inputToken[tokenNumber].tokenType = mulop;

else if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))

inputToken[tokenNumber].tokenType = id;

else

inputToken[tokenNumber].tokenType = undefine;

tokenNumber++;

}

inputToken[tokenNumber++].tokenType = endInput;

}

TreeNode\* E() {

TreeNode\* node = new TreeNode;

node->flag = "E";

switch (inputToken[curPos].tokenType)

{

case id:

case leftBracket: {

TreeNode\* tNode = node->firstChild = T();

if (error)

return node;

tNode = tNode->nextSibling = E1();

break;

}

default:

node = errorNode();

error = true;

break;

}

return node;

}

TreeNode\* E1() {

TreeNode\* node = new TreeNode;

node->flag = "E'";

switch (inputToken[curPos].tokenType)

{

case endInput:

case rightBracket:

(node->firstChild = new TreeNode)->flag = "empty";

break;

case addop: {

TreeNode\* tNode = node->firstChild = match();

tNode = tNode->nextSibling = T();

if (error)

return node;

tNode = tNode->nextSibling = E1();

break;

}

default:

node = errorNode();

error = true;

break;

}

return node;

}

TreeNode\* T() {

TreeNode\* node = new TreeNode;

node->flag = "T";

switch (inputToken[curPos].tokenType)

{

case id:

case leftBracket: {

TreeNode\* tNode = node->firstChild = F();

if (error)

return node;

tNode = tNode->nextSibling = T1();

break;

}

default:

node = errorNode();

error = true;

break;

}

return node;

}

TreeNode\* T1() {

TreeNode\* node = new TreeNode;

node->flag = "T'";

switch (inputToken[curPos].tokenType)

{

case endInput:

case rightBracket:

case addop:

(node->firstChild = new TreeNode)->flag = "empty";

break;

case mulop: {

TreeNode\* tNode = node->firstChild = match();

tNode = tNode->nextSibling = F();

if (error)

return node;

tNode = tNode->nextSibling = T1();

break;

}

default:

node = errorNode();

error = true;

break;

}

return node;

}

TreeNode\* F() {

TreeNode\* node = new TreeNode;

node->flag = "F";

switch (inputToken[curPos].tokenType)

{

case id:

node->firstChild = match();

break;

case leftBracket: {

TreeNode\* tNode = node->firstChild = match();

tNode = tNode->nextSibling = E();

if (error)

return node;

tNode = tNode->nextSibling = match();

break;

}

default:

node = errorNode();

error = true;

break;

}

return node;

}

TreeNode\* match() {

TreeNode\* newNode = new TreeNode;

newNode->flag = "match";

newNode->firstChild = newNode->nextSibling = NULL;

newNode->token = &(inputToken[curPos++]);

return newNode;

}

TreeNode\* errorNode() {

TreeNode\* errorNode = new TreeNode;

errorNode->flag = "error";

errorNode->firstChild = errorNode->nextSibling = NULL;

errorNode->token = &(inputToken[curPos++]);

return errorNode;

}

void printTree(TreeNode\* head, int depth /\*defalut=0\*/, bool isFirst /\*default=false\*/) {

if (head == NULL)

return;

int i;

for (i = 0; i < depth - 1; i++)

cout << "\t |";

if (i < depth)

cout << (isFirst ? "\t↖" : "\t |");

if (head->flag == "match") {

MyTokenType type = head->token->tokenType;

string value = head->token->value;

cout << typeName[type] << ':';

SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_GREEN);

cout << value << endl;

SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

}

else

{

if (head->flag == "error")

SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED);

cout << head->flag << endl;

SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

printTree(head->firstChild, depth + 1, true);

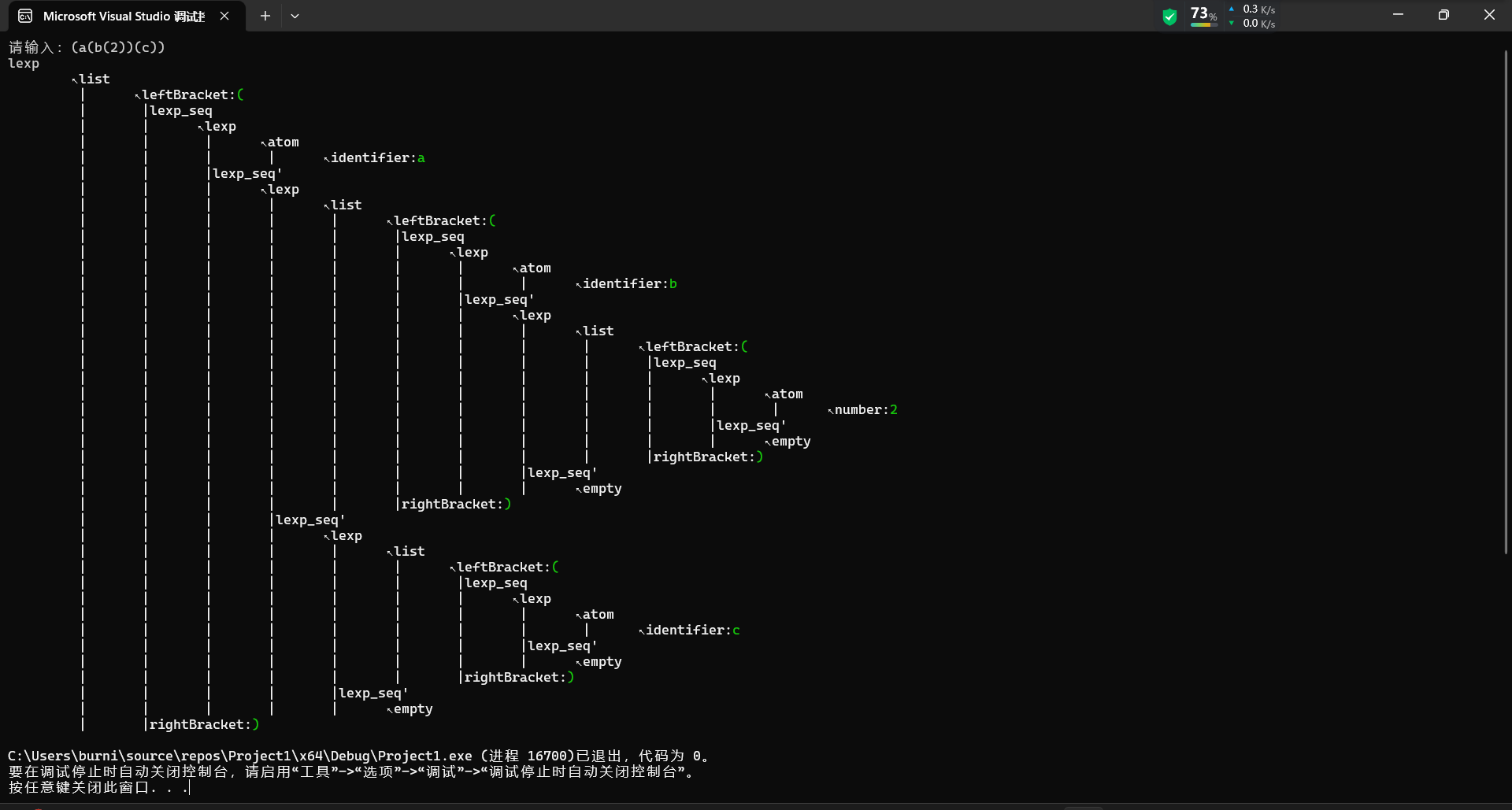
}

printTree(head->nextSibling, depth);

}

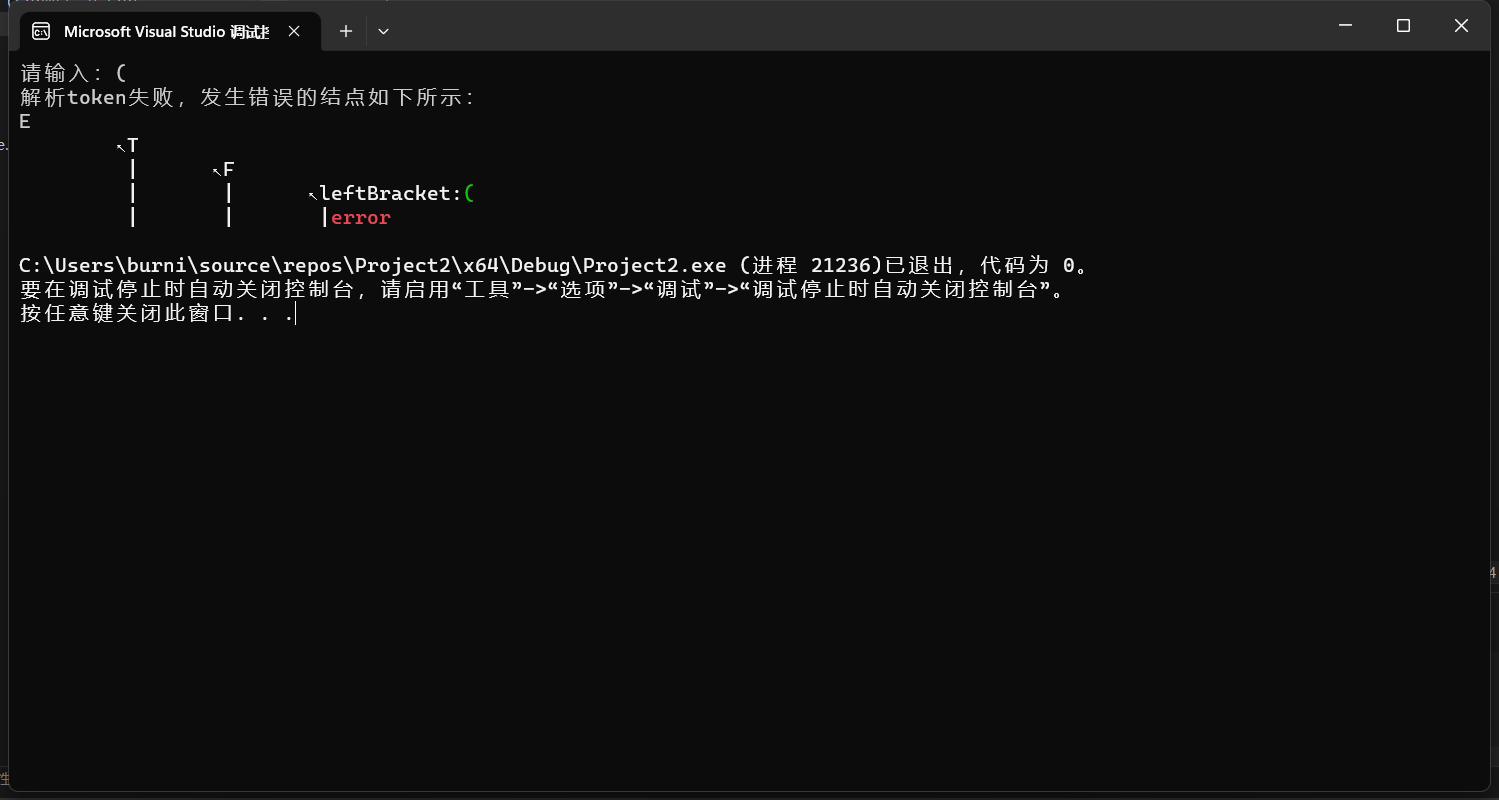
1. 截图

* 习题4.8输出截图



* 给定文法输出截图





匹配错误

1. 实验结论:

1 、实验结论

（是否能够准确描述实验的结论）

自顶向下的分析算法通过在最左推导中描述出各个步骤来分析记号串输入。预测分析程序试图利用一个或多个先行记号来预测出输入串中的下一个构造，而回溯分析程序则试着分析其他可能的输入，当一种可能失败时就要求输入中备份任意数量的字符。

  实际上，待编译的文本首先通过词法分析器，分解得到Token序列，再对Token进行语法分析，得到分析树。因此，语法分析算法部分的处理对象是Token结构体，而不是简单的字符序列。但由于本次实验的输入较为简单，且侧重点在于语法分析，因此仅对输入进行了简单的处理（见输入设计）。但是，只需要对源代码作简单的修改，将input函数修改为对输入字符串进行词法分析的函数（结合实验一的内容），即可更为真实地模拟出编译器的执行过程。

2、分析和总结

1）对输入设计的结论

严格来说，输入的字符串首先通过词法分析得到Token序列，再经过本算法得到分析树。但是本次实验的输入较为简单，且侧重点在于后者，故仅对输入串进行简单的判断，生成Token序列。input函数把输入字符串中的每个字符视作一个Token。一个字母即是一个identifier，一个数字即为一个number。

2）对输出设计的结论

算法的输出为语法分析树（用树的孩子兄弟表示法，存储结构为二叉树）。打印树时，以缩进数代表结点的深度。

3）对LL(1)分析法的结论

递归下降分析算法需要消除左递归和提取左因子。由于递归下降分析算法按照最左推导递归调用函数，若存在左递归将面临无限递归下去的局面，若存在相同左因子的选择则面临选择不同的推导的问题。

3、对预估问题的结论

LL(1)文法一定是无二义性文法，若给定文法是二义性文法，则无法通过LL(1)分析过程得到分析树。若给定文法不是二义性文法，但存在左递归或左因子，则要消除左递归和左因子，再构造出LL(1)分析表。