编译器专题实验报告

**实验六、七：符号表与可执行代码构建（独立模式）**

**一、实验目的**

实验六：

目的：设计和实现一个符号表。

符号表生成阶段：

词法分析：遇到一个新声明的变量名、函数名，添加到符号表；

语法分析：填充符号表的相关信息；

语义分析：依据符号表作语义正确性检查代码及测试。

实验七：

目的：将语义分析输出的符号表映射为内存映像，并生成依赖于栈帧的目标代码，将结果输出到文件中。

功能：

栈帧设计（含D表）；

序言、尾声、调用序列、返回序列构建；

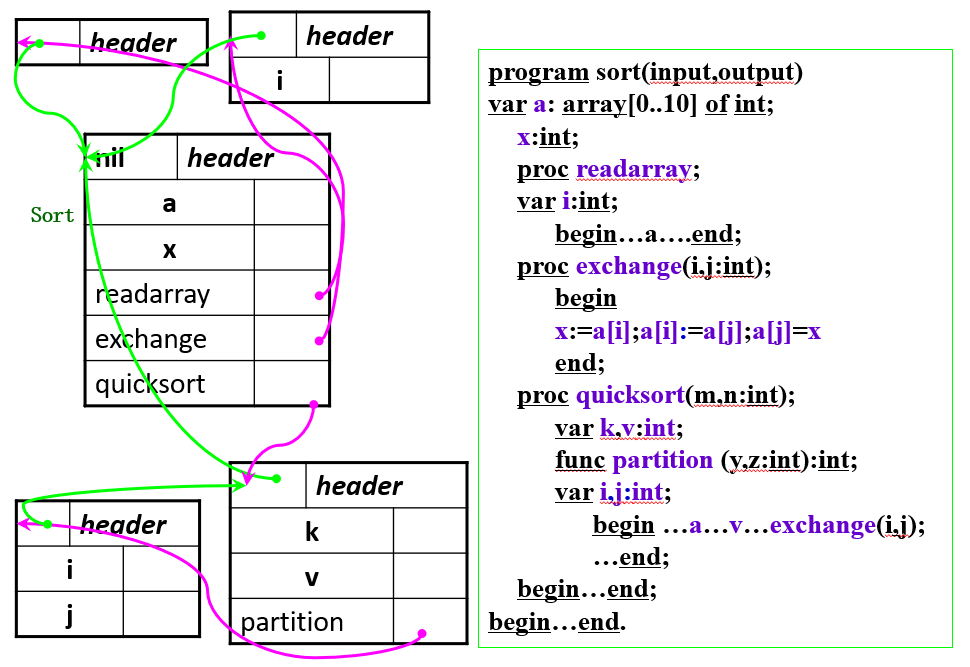
名引用的代码变换（引用序列构建）；

目标语言指令模板（MIPS）

**二、实验一：符号表**

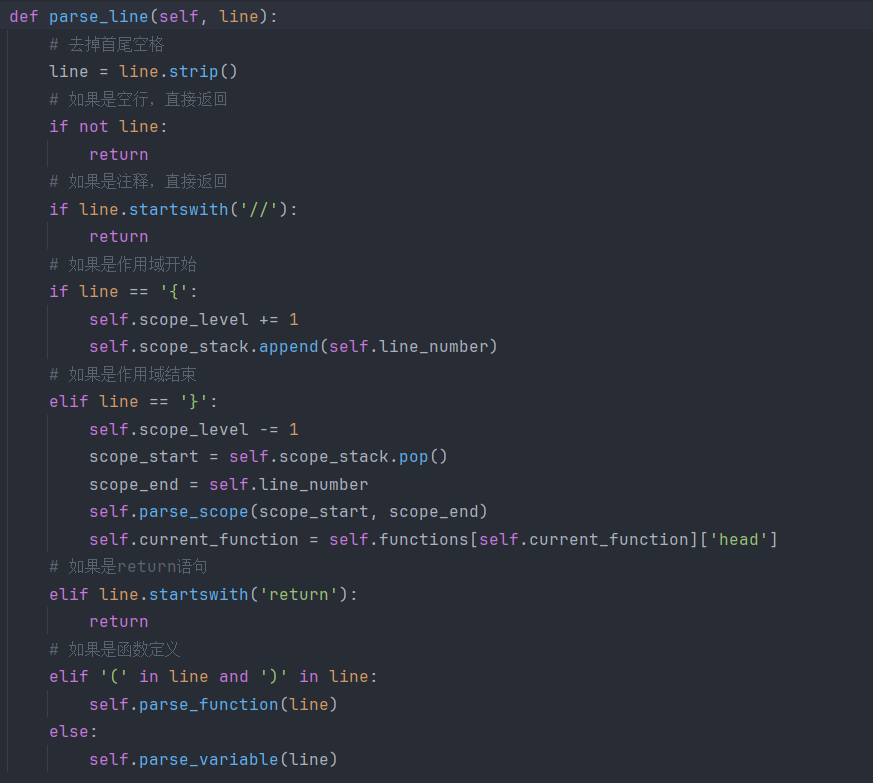
**1. 符号表设计**

为每个函数开辟一张符号表，并将各个函数连接起来，如下图所示。

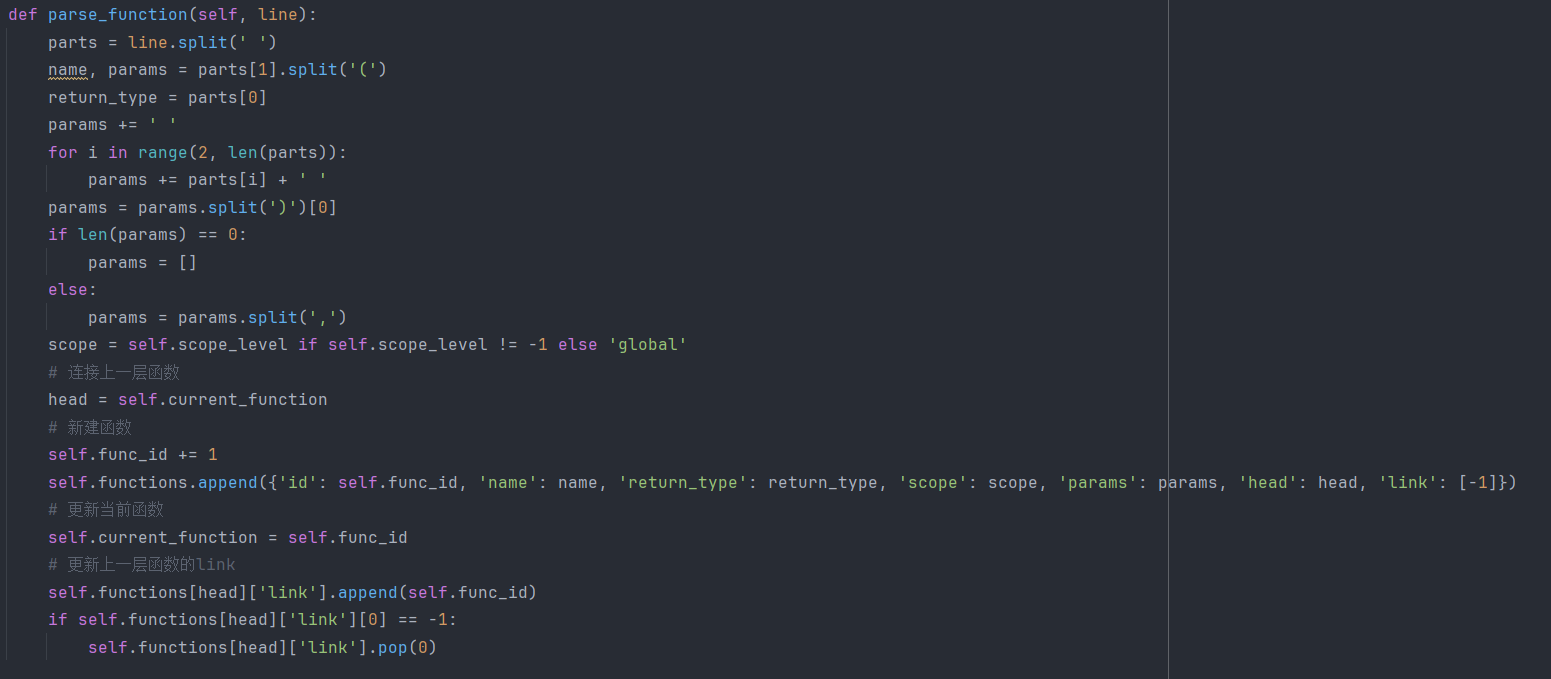


**2. 程序设计**

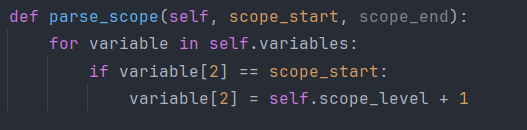
使用Python实现符号表的设计，程序首先读入文本文件中的程序，并根据读入程序的内容识别该行语句是否为函数或者变量声明语句，根据读入的大括号进行作用域的跟踪，使用scope level变量实现，同时使用current function变量表示当前在哪个函数中，用于不同函数符号表之间的连接，主要函数如下所示，根据读入的内容进行不同的操作。



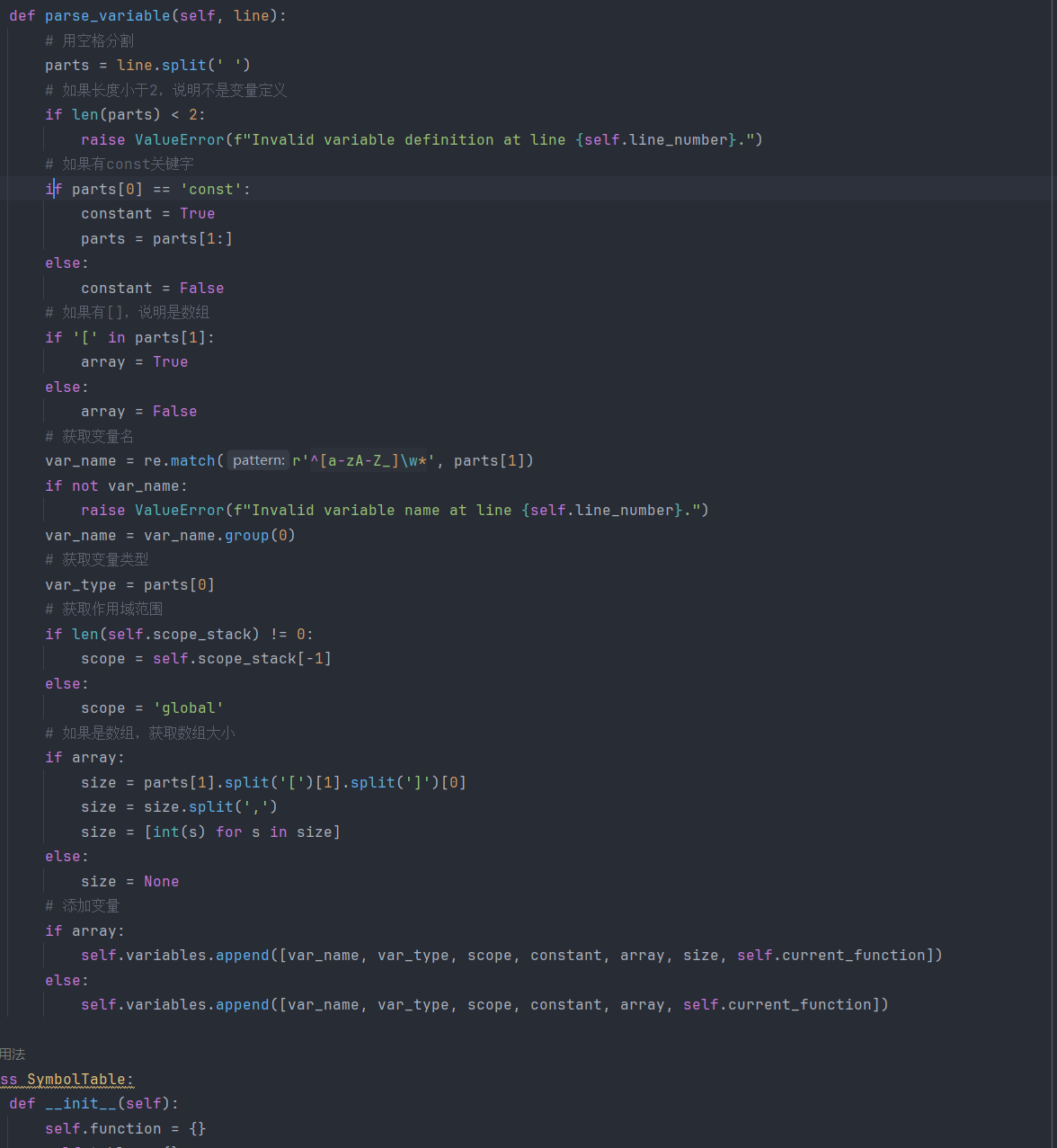
当发现读入函数声明时，获取函数的名称，返回类型，参数表等，并记录外层函数，即header的值，主要代码如下。



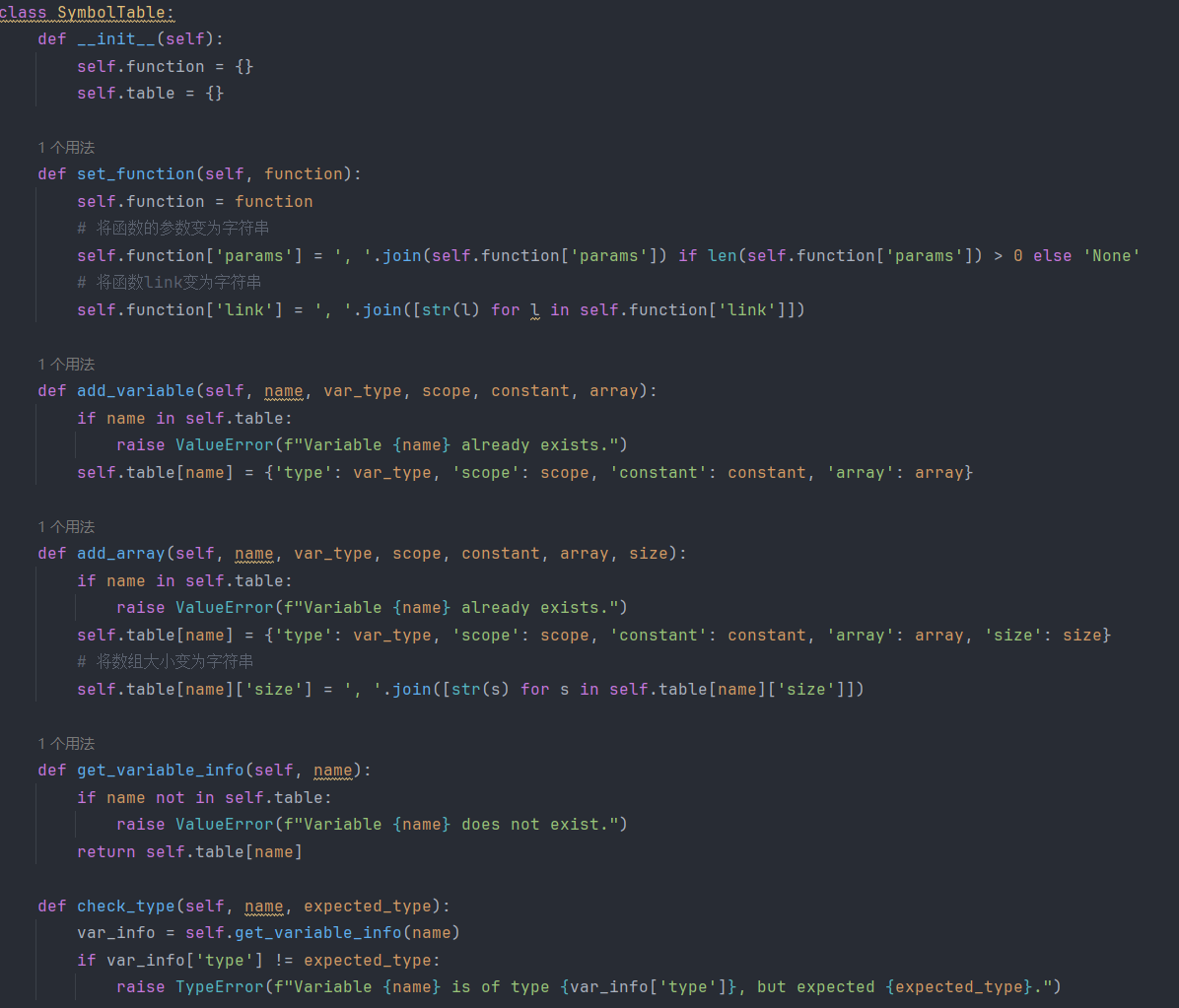
当发现读入大括号时，需要进行作用域的操作，根据大括号读入，确定变量或者函数的起始和终止范围，并确定作用域的级别，这里认为全局变量或者函数的scope为global，在全局函数内部定义的变量的scope等级为0，再在内部定义的scope即为1,2,3...。



同时还需要对读入的变量进行分析，判断是否为数组或者常变量，并得到变量名，变量类型等信息，如果是数组还需要统计数组的长度等信息。

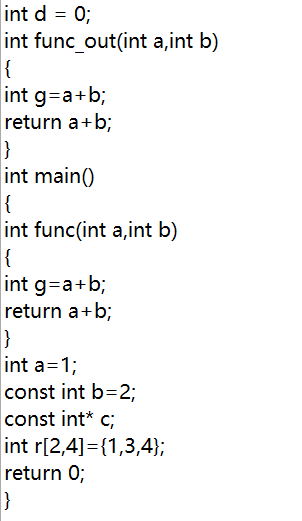


根据上面得到的变量和函数的信息，使用符号表类进行符号表的填写，判断是否有符号名或者变量名的重复，为每个函数填写一个符号表，最终得到整个程序的符号表。

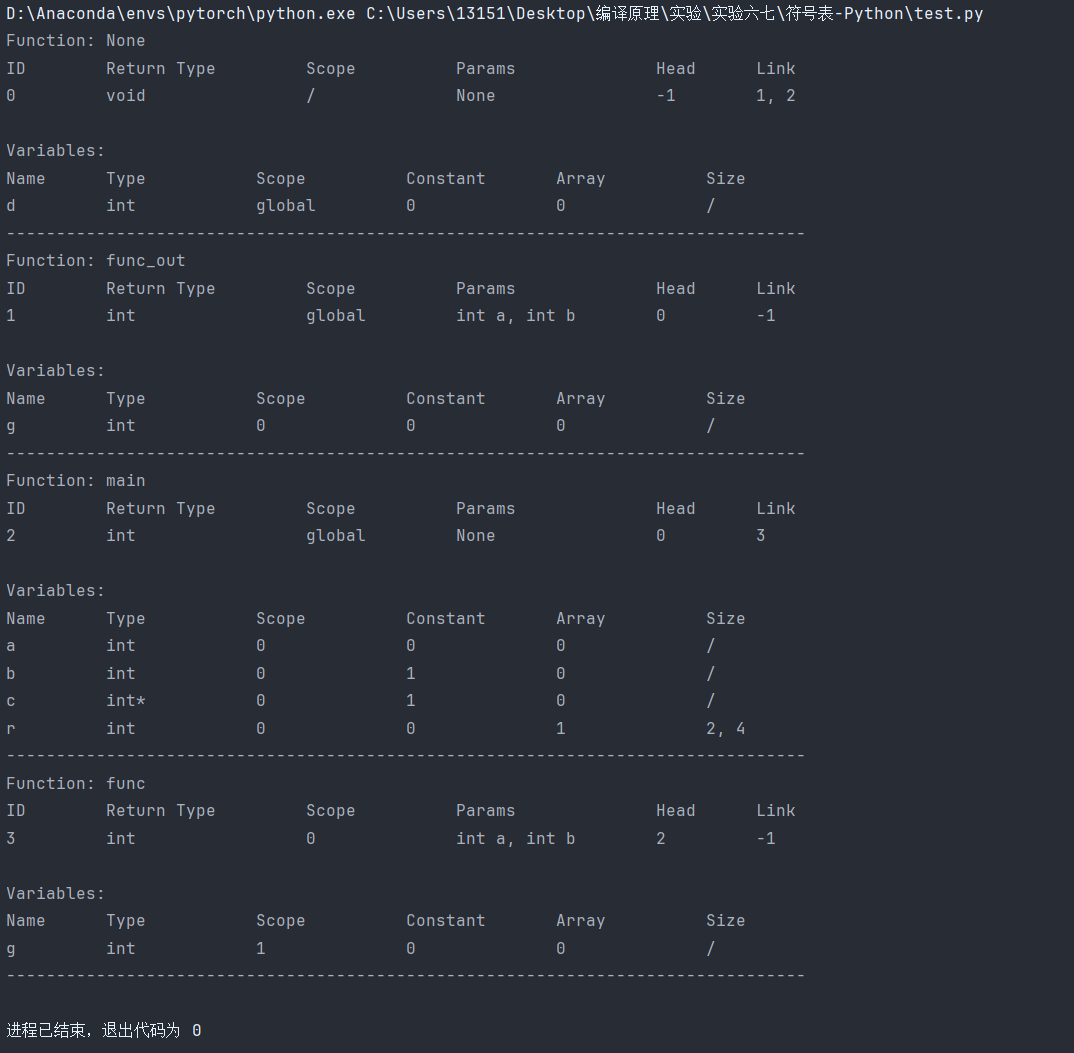


**3. 实验验证**

对以下的程序构建符号表。



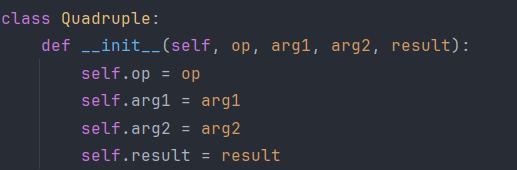
可以得到如下的结果，其中函数名为None的是最外层的无名函数，它没有header，但是连接着两个定义的全局函数，一个是ID为1的func\_out，另外是一个ID为2的main函数。main函数中又有定义函数func，所以main函数一方面header为ID为0的无名函数，另一方面连接着ID为3的func，同理func的header为mian，由于其内层没有定义函数，所以link为-1。除此之外，每个函数内部还有定义变量，无名函数内部的变量为全局变量。mian函数中还有定义int类型的数组，大小为2×4。



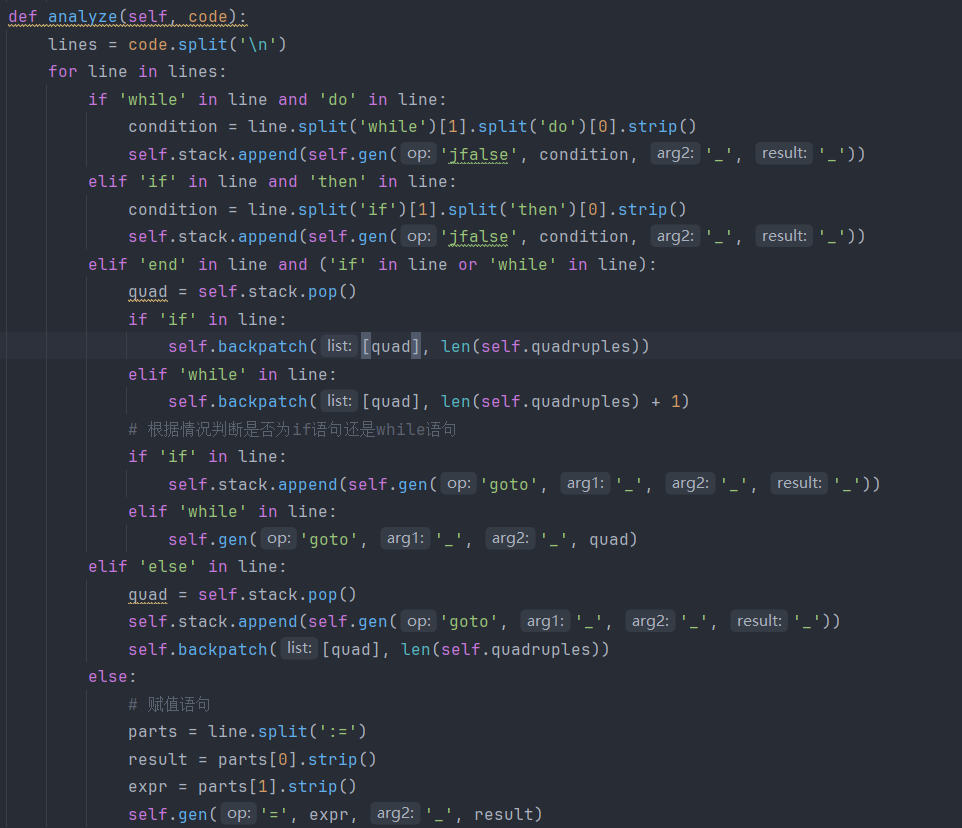
**三、实验二：拉链回填**

**1. 程序设计**

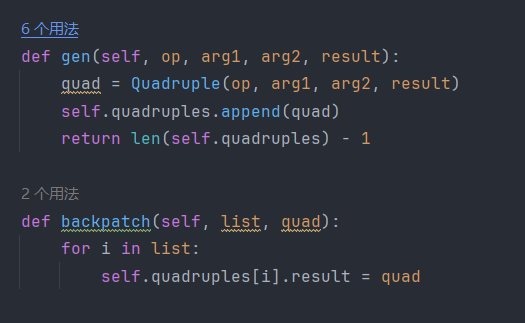
对if while等控制语句进行中间代码的四元式生成，使用拉链反填的方法，首先定义四元式类，如下所示。



读入文本文件中的代码，根据读入的内容编写不同的操作，如下图所示，其中四元式的地址部分大多没有具体确定，需要使用拉链反填技术。

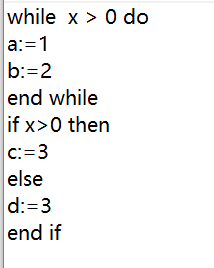


当读入到诸如end，else等字符串的时候使用如下所示的拉链反填函数和四元式生成函数，填写空缺的四元式地址内容。

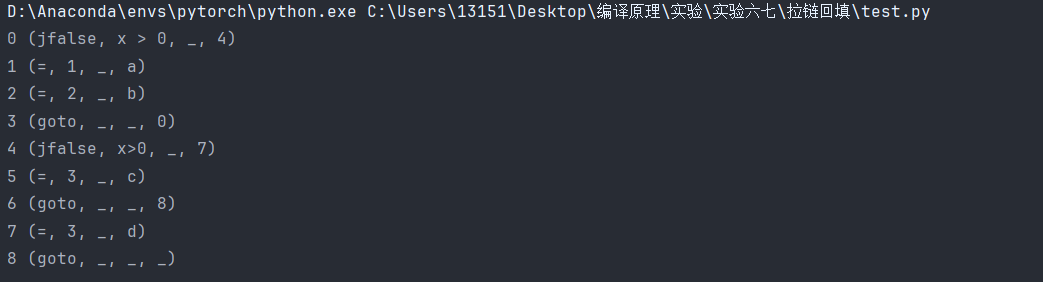


**2. 程序验证**

令程序读入以下的程序内容，进行带有拉链反填的四元式生成。



得到生成的四元式代码如下，可以看到程序能够将if和while的控制语句均有效地翻译为四元式。



**四、实验三：可执行代码构建**

**1. 使用文法**

PROG   → { DECLS STMTS  }

DECLS  → DECLS DECL | empty

DECL   → int  NAMES  ; | bool  NAMES  ;

NAMES  → NAMES , NAME  | NAME

NAME   → id

STMTS  → STMTS STMT   | STMT

STMT   → id  = EXPR ; | id: = BOOL;

STMT   → if id  then STMT

STMT   → if  id  then STMT  else STMT

STMT   → while  id  do STMT

STMT   → { STMTS  STMT   }

STMT   → read id ;

STMT   → write id ;

EXPR   → EXPR ADD TERM   | TERM

ADD    → + | -

TERM   → TERM MULNEGA  | NEGA

MUL    → \* |/

NEGA   → FACTOR  | -FACTOR

FACTOR → (EXPR ) | id  | number

BOOL  → BOOL  || JOIN  | JOIN

JOIN   → JOIN  && NOT   | NOT

NOT    → REL  | !REL

REL    → EXPR  ROP EXPR

ROP    → > | >= | < | <= | == |  !=。

在改写成LL1文法之后：  
PROG → DECLS STMTS

DECLS → DECL DECLS' | ε

DECLS' → DECL DECLS' | ε

DECL → int NAMES ; | bool NAMES ;

NAMES → NAME NAMES'

NAMES' → , NAME NAMES' | ε

NAME → id

STMTS → STMT STMTS' | ε

STMTS' → STMT STMTS' | ε

STMT → id STMT' | = EXPR; | := BOOL; | if id then STMT ELSESTMT | while id do STMT | { STMTS } | read id ; | write id ;

ELSESTMT → else STMT | ε

EXPR → TERM ADD'

ADD' → ADD TERM ADD' | ε

ADD → + | -

TERM → NEGA MUL'

MUL' → MUL NEGA MUL' | ε

MUL → \* | /

NEGA → FACTOR | - FACTOR

FACTOR → ( EXPR ) | id | number

BOOL → JOIN BOOL'

BOOL' → || JOIN BOOL' | ε

JOIN → NOT JOIN'

JOIN' → && NOT JOIN' | ε

NOT → REL | ! REL

REL → EXPR ROP EXPR

ROP → > | >= | < | <= | == | !=

**2. 程序主体**

使用递归下降的方式来编写中间代码生成这个程序，为每一个非终结符都定义了一个处理函数，如下所示。

void processPROG(TreeNode\* node);

void processPROG(TreeNode\* node);

void processDECLS(TreeNode\* node);

void processDECLS\_(TreeNode\* node);

void processDECL(TreeNode\* node);

vector<string> processNAMES(TreeNode\* node);

vector<string> processNAMES\_(TreeNode\* node);

void processSTMTS(TreeNode\* node);

void processSTMTS\_(TreeNode\* node);

void processSTMT(TreeNode\* node);

void processSTMT\_(TreeNode\* root, string varName);

void processIfStmt(TreeNode\* node);

void processELSESTMT(TreeNode\* node);

void processWhileStmt(TreeNode\* node);

void processReadStmt(TreeNode\* node);

void processWriteStmt(TreeNode\* node);

void processEXPR(TreeNode\* node);

void processADD\_(TreeNode\* node);

void processTERM(TreeNode\* node);

void processMUL\_(TreeNode\* node);

void processNEGA(TreeNode\* node);

void processFACTOR(TreeNode\* node);

void processBOOL(TreeNode\* node);

void processBOOL\_(TreeNode\* node);

void processJOIN(TreeNode\* node);

void processJOIN\_(TreeNode\* node);

void processNOT(TreeNode\* node);

void processREL(TreeNode\* node);

使用符号表处理变量声明和类型检查

vector<pair<pair<string,string>,int>>symboltable;

三个值分别存放名字，类型和初始值。如果声明时未初始化则设为-1

在递归下降结束之后，遍历符号表，将符号表中的所有变量对应的声明写在目标汇编语言的.data部分。然后由于代码生成时汇编代码部分很繁琐且重复。我为要重复用到的mips汇编代码写了多个生成函数，如下所示。

void int\_stack\_init() {

emit("#int栈初始化,t8为int栈指针");

emit("la $t8, int\_stack");

emit("addiu $t8, $t8, 508");

}

//t1入栈

void int\_stack\_push() {

emit("#t1入栈");

emit("addiu $t8, $t8, -4");

emit("sw $t1, 0($t8)");

}

//int出栈

void int\_stack\_pop\_1() {

emit("#出栈到t1");

emit("lw $t1, 0($t8)");

emit("addiu $t8, $t8, 4");

}

void int\_stack\_pop\_1\_2() {

emit("#出栈到t1与t2");

emit("lw $t1, 0($t8)");

emit("addiu $t8, $t8, 4");

emit("lw $t2, 0($t8)");

emit("addiu $t8, $t8, 4");

}

void bool\_stack\_init() {

emit("#bool栈初始化");

emit("la $t9, bool\_stack");

emit("addiu $t9, $t9, 127");

}

void bool\_stack\_push\_1() {

emit("#t1入栈");

emit("addiu $t9, $t9, -1");

emit("sb $t1, 0($t9)");

}

void bool\_stack\_pop\_1() {

emit("#出栈到t1");

emit("lb $t1, 0($t9)");

emit("addiu $t9, $t9, 1");

}

void bool\_stack\_pop\_1\_2() {

emit("#出栈到t1和t2");

emit("lb $t1, 0($t9)");

emit("addiu $t9, $t9, 1");

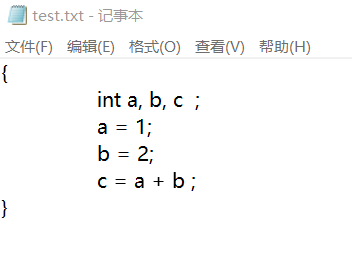
emit("lb $t2, 0($t9)");

emit("addiu $t9, $t9, 1");

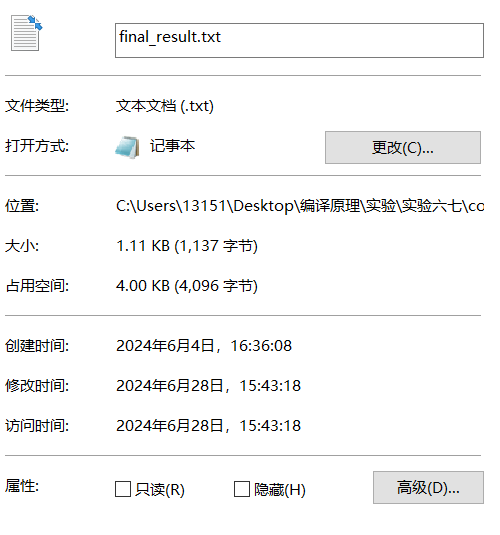
}

**3. 验证实例**

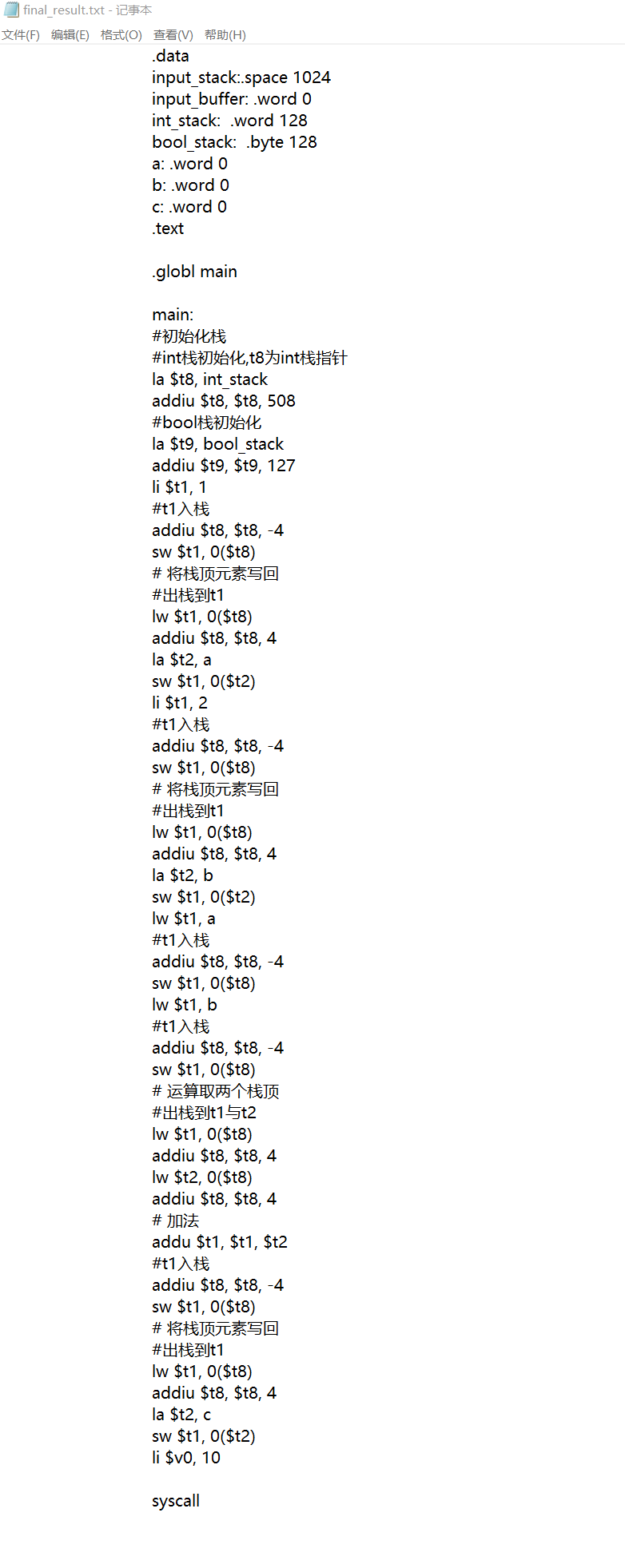
在test.txt文件中写入需要进行编译的程序如下所示。



运行编译程序之后得到final\_result.txt文件为MIPS代码，文件信息如下所示。



其中的MIPS代码如下所示。



能够实现程序需要的功能。

**五、遇到的问题**

汇编代码生成和编译器的后端

https://blog.csdn.net/lijj0304/article/details/135567752

**六、代码**

1. 符号表

import re

class Parser:

def \_\_init\_\_(self, code):

self.code = code

self.line\_number = 0

self.scope\_stack = []

self.scope\_level = -1

self.variables = []

self.functions = [{'id': 0, 'name': 'None', 'return\_type': 'void', 'scope': '/', 'params': [], 'head': -1, 'link': [-1]}]

self.current\_function = 0

self.func\_id = 0

def parse(self):

lines = self.code.split('\n')

for line in lines:

self.line\_number += 1

self.parse\_line(line)

# 将func id对应的function和variable关联起来

func\_vars = []

for func in self.functions:

func\_var = {'function': func, 'variables': []}

for var in self.variables:

if var[-1] == func['id']:

func\_var['variables'].append(var)

func\_vars.append(func\_var)

return func\_vars

def parse\_line(self, line):

# 去掉首尾空格

line = line.strip()

# 如果是空行，直接返回

if not line:

return

# 如果是注释，直接返回

if line.startswith('//'):

return

# 如果是作用域开始

if line == '{':

self.scope\_level += 1

self.scope\_stack.append(self.line\_number)

# 如果是作用域结束

elif line == '}':

self.scope\_level -= 1

scope\_start = self.scope\_stack.pop()

scope\_end = self.line\_number

self.parse\_scope(scope\_start, scope\_end)

self.current\_function = self.functions[self.current\_function]['head']

# 如果是return语句

elif line.startswith('return'):

return

# 如果是函数定义

elif '(' in line and ')' in line:

self.parse\_function(line)

else:

self.parse\_variable(line)

def parse\_function(self, line):

parts = line.split(' ')

name, params = parts[1].split('(')

return\_type = parts[0]

params += ' '

for i in range(2, len(parts)):

params += parts[i] + ' '

params = params.split(')')[0]

if len(params) == 0:

params = []

else:

params = params.split(',')

scope = self.scope\_level if self.scope\_level != -1 else 'global'

# 连接上一层函数

head = self.current\_function

# 新建函数

self.func\_id += 1

self.functions.append({'id': self.func\_id, 'name': name, 'return\_type': return\_type, 'scope': scope, 'params': params, 'head': head, 'link': [-1]})

# 更新当前函数

self.current\_function = self.func\_id

# 更新上一层函数的link

self.functions[head]['link'].append(self.func\_id)

if self.functions[head]['link'][0] == -1:

self.functions[head]['link'].pop(0)

def parse\_scope(self, scope\_start, scope\_end):

for variable in self.variables:

if variable[2] == scope\_start:

variable[2] = self.scope\_level + 1

def parse\_variable(self, line):

# 用空格分割

parts = line.split(' ')

# 如果长度小于2，说明不是变量定义

if len(parts) < 2:

raise ValueError(f"Invalid variable definition at line {self.line\_number}.")

# 如果有const关键字

if parts[0] == 'const':

constant = True

parts = parts[1:]

else:

constant = False

# 如果有[]，说明是数组

if '[' in parts[1]:

array = True

else:

array = False

# 获取变量名

var\_name = re.match(r'^[a-zA-Z\_]\w\*', parts[1])

if not var\_name:

raise ValueError(f"Invalid variable name at line {self.line\_number}.")

var\_name = var\_name.group(0)

# 获取变量类型

var\_type = parts[0]

# 获取作用域范围

if len(self.scope\_stack) != 0:

scope = self.scope\_stack[-1]

else:

scope = 'global'

# 如果是数组，获取数组大小

if array:

size = parts[1].split('[')[1].split(']')[0]

size = size.split(',')

size = [int(s) for s in size]

else:

size = None

# 添加变量

if array:

self.variables.append([var\_name, var\_type, scope, constant, array, size, self.current\_function])

else:

self.variables.append([var\_name, var\_type, scope, constant, array, self.current\_function])

class SymbolTable:

def \_\_init\_\_(self):

self.function = {}

self.table = {}

def set\_function(self, function):

self.function = function

# 将函数的参数变为字符串

self.function['params'] = ', '.join(self.function['params']) if len(self.function['params']) > 0 else 'None'

# 将函数link变为字符串

self.function['link'] = ', '.join([str(l) for l in self.function['link']])

def add\_variable(self, name, var\_type, scope, constant, array):

if name in self.table:

raise ValueError(f"Variable {name} already exists.")

self.table[name] = {'type': var\_type, 'scope': scope, 'constant': constant, 'array': array}

def add\_array(self, name, var\_type, scope, constant, array, size):

if name in self.table:

raise ValueError(f"Variable {name} already exists.")

self.table[name] = {'type': var\_type, 'scope': scope, 'constant': constant, 'array': array, 'size': size}

# 将数组大小变为字符串

self.table[name]['size'] = ', '.join([str(s) for s in self.table[name]['size']])

def get\_variable\_info(self, name):

if name not in self.table:

raise ValueError(f"Variable {name} does not exist.")

return self.table[name]

def check\_type(self, name, expected\_type):

var\_info = self.get\_variable\_info(name)

if var\_info['type'] != expected\_type:

raise TypeError(f"Variable {name} is of type {var\_info['type']}, but expected {expected\_type}.")

# Read code from file

with open('code.txt', 'r') as file:

code = file.read()

# Parse the code

parser = Parser(code)

func\_vars = parser.parse()

# Add variables to symbol table

symbol\_tables = []

for func\_var in func\_vars:

symbol\_table = SymbolTable()

symbol\_table.set\_function(func\_var['function'])

for var in func\_var['variables']:

if var[4]:

symbol\_table.add\_array(var[0], var[1], var[2], var[3], var[4], var[5])

else:

symbol\_table.add\_variable(var[0], var[1], var[2], var[3], var[4])

symbol\_tables.append(symbol\_table)

# 检查重复的函数名

function\_names = [symbol\_table.function['name'] for symbol\_table in symbol\_tables]

if len(function\_names) != len(set(function\_names)):

raise ValueError("Duplicate function names.")

# Get variable info

for symbol\_table in symbol\_tables:

print('Function:', symbol\_table.function['name'])

# 输出函数信息：ID, Return Type, Scope, Params, Head, Link，分别占用10，20，15，20，10，10个字符，左对齐

print(f"{'ID':<10}{'Return Type':<20}{'Scope':<15}{'Params':<20}{'Head':<10}{'Link':<10}")

print(f"{symbol\_table.function['id']:<10}{symbol\_table.function['return\_type']:<20}{symbol\_table.function['scope']:<15}{symbol\_table.function['params']:<20}{symbol\_table.function['head']:<10}{symbol\_table.function['link']:<10}")

print('\nVariables:')

# 输出变量信息：Name, Type, Scope, Constant, Array, Size，分别占用10，15，15，15，15，10个字符，左对齐

print(f"{'Name':<10}{'Type':<15}{'Scope':<15}{'Constant':<15}{'Array':<15}{'Size':<10}")

for name, info in symbol\_table.table.items():

if info['array']:

print(f"{name:<10}{info['type']:<15}{info['scope']:<15}{info['constant']:<15}{info['array']:<15}{info['size']:<10}")

else:

print(f"{name:<10}{info['type']:<15}{info['scope']:<15}{info['constant']:<15}{info['array']:<15}{'/':<10}")

# 输出分割线

print('-' \* 80)

2. 拉链回填

class Quadruple:

def \_\_init\_\_(self, op, arg1, arg2, result):

self.op = op

self.arg1 = arg1

self.arg2 = arg2

self.result = result

class SemanticAnalyzer:

def \_\_init\_\_(self):

self.quadruples = []

self.stack = []

def gen(self, op, arg1, arg2, result):

quad = Quadruple(op, arg1, arg2, result)

self.quadruples.append(quad)

return len(self.quadruples) - 1

def backpatch(self, list, quad):

for i in list:

self.quadruples[i].result = quad

def analyze(self, code):

lines = code.split('\n')

for line in lines:

if 'while' in line and 'do' in line:

condition = line.split('while')[1].split('do')[0].strip()

self.stack.append(self.gen('jfalse', condition, '\_', '\_'))

elif 'if' in line and 'then' in line:

condition = line.split('if')[1].split('then')[0].strip()

self.stack.append(self.gen('jfalse', condition, '\_', '\_'))

elif 'end' in line and ('if' in line or 'while' in line):

quad = self.stack.pop()

if 'if' in line:

self.backpatch([quad], len(self.quadruples))

elif 'while' in line:

self.backpatch([quad], len(self.quadruples) + 1)

# 根据情况判断是否为if语句还是while语句

if 'if' in line:

self.stack.append(self.gen('goto', '\_', '\_', '\_'))

elif 'while' in line:

self.gen('goto', '\_', '\_', quad)

elif 'else' in line:

quad = self.stack.pop()

self.stack.append(self.gen('goto', '\_', '\_', '\_'))

self.backpatch([quad], len(self.quadruples))

else:

# 赋值语句

parts = line.split(':=')

result = parts[0].strip()

expr = parts[1].strip()

self.gen('=', expr, '\_', result)

with open('code.txt', 'r') as f:

code = f.read()

analyzer = SemanticAnalyzer()

analyzer.analyze(code)

start = 0

for quad in analyzer.quadruples:

print(f'{start} ({quad.op}, {quad.arg1}, {quad.arg2}, {quad.result})')

start += 1

3.代码生成

#include"lexical\_analyzer.h"

#include"Parser.h"

#include"getx86.h"

int main() {

string a = "C:\\Users\\13151\\Desktop\\编译原理\\实验\\实验六七\\compile\\compile-main\\test.txt"; //源代码

string b = "C:\\Users\\13151\\Desktop\\编译原理\\实验\\实验六七\\compile\\compile-main\\result.txt"; //词法分析结果

string e = "C:\\Users\\13151\\Desktop\\编译原理\\实验\\实验六七\\compile\\compile-main\\result\_parser.txt"; //语法分析结果

string test = "test\_parser\_result.txt"; //根据语法分析结果重新构造树之后再进行序列化的结果

Parser d(a, b, e);

if (d.do\_and\_write()) {

TreeNode\* root = d.deserialize(e);//反序列化构造树

d.serialize(test, root); //序列化反序列化结果

if (file\_same(e, test))

cout << "反序列化成功" << endl; //判断两个序列化结果是否一致

else

cout << "反序列化失败" << endl;

processPROG(root);

}

return 0;

}

#pragma once

#pragma once

#include<sstream>

#include <vector>

#include<iostream>

#include<fstream>

#include<string>

#include<unordered\_map>

//本程序实现的是一个叫little\_c的语言

/\*a. 该语言的一个程序由且仅由一个块结构组成，该块中包含两部分：可选的声明语句和必须的执行语句。声明语句用于声明变量，执行语句用于表达计算过程，过程中使用这些变量来存储内部数据和计算结果。

b. 该语言支持两种数据类型：整型和布尔型。整型数据占4个字节，布尔型占1个字节。整型数据的常量是整常数，布尔型数据的常量是 true 和 false 。

c. 整型数据支持 +、-、\*、/ 四种算术运算，运算结果为整型值。整型数据可与>、>=...等六个关系运算符结合，运算结果为布尔型值。

d. 布尔型数据支持 || 、&& 、！三种逻辑运算，运算结果为布尔常量。

e. 整型的变量和常量可以用算术运算符组合成算术表达式，其值为整常数；布尔型的变量和常量可以用逻辑运算符组成成布尔表达式，其值为布尔常量；算术表达式可以加上关系运算符构造出布尔表达式。

f. 整常数和算术表达式可以用于整型变量赋值，布尔常量和布尔表达式可以用于布尔型变量赋值；两者不兼容（文法中使用不同的赋值运算符以示区别）。

g. 各种运算符的优先级和结合规则参照C语言语法理解。

h. 控制流语句中的控制条件限定为1个布尔型变量，该量应该在引用之前定值。

i. 该语言不支持数组、结构体、指针等复杂数据类型。

j. 该语言不含有子程序结构，也不支持过程调用。

k. 该语言有简单的I/O功能：使用read指令能够从键盘读入1个整常数，使用write指令能向屏幕打印输出1个整常数。\*/

/\*文法定义：

PROG → { DECLS STMTS }

DECLS → DECLS DECL | empty

DECL → int NAMES ; | bool NAMES ;

NAMES → NAMES , NAME | NAME

NAME → id

STMTS → STMTS STMT | STMT

STMT → id = EXPR ; | id := BOOL ;

STMT → if id then STMT

STMT → if id then STMT else STMT

STMT → while id do STMT

STMT → { STMTS STMT }

STMT → read id ;

STMT → write id ;

EXPR → EXPR ADD TERM | TERM

ADD → + | -

TERM → TERM MUL NEGA | NEGA

MUL → \* | /

NEGA → FACTOR | - FACTOR

FACTOR→ ( EXPR ) | id | number

BOOL → BOOL || JOIN | JOIN

JOIN → JOIN && NOT | NOT

NOT → REL | ! REL

REL → EXPR ROP EXPR

ROP → > | >= | < | <= | == | !=\*/

using namespace std;

enum Token {

IDENTIFIER, //标识符

NUMBER, //整常数

BOOL, //布尔常量

KEYWORD, //关键字

OPERATOR, //运算符

SEPARATOR, //分隔符

INVALID\_TOKEN, //无效词

};

class lexical\_analyzer {

public:

lexical\_analyzer(const string& input\_file\_name, const string& output\_file\_name);

~lexical\_analyzer();

std::vector<pair<string, string>> result\_to\_token(const string& ifn);

bool analyze();

private:

unordered\_map<string, Token> keyword\_map = {

{"int",KEYWORD},

{"bool",KEYWORD},

{"if",KEYWORD},

{"then",KEYWORD},

{"while",KEYWORD},

{"do",KEYWORD},

{"read",KEYWORD},

{"write",KEYWORD},

{"else",KEYWORD},

};

unordered\_map<string, Token> operator\_map = {

{"+", OPERATOR},

{"-", OPERATOR},

{"\*", OPERATOR},

{"/", OPERATOR},

{"||", OPERATOR},

{"&&", OPERATOR},

{"!", OPERATOR},

{">=", OPERATOR},

{">", OPERATOR},

{"<=", OPERATOR},

{"<", OPERATOR},

{"==", OPERATOR},

{":=", OPERATOR},

{"=", OPERATOR},

};

unordered\_map<string, Token> separator\_map = {

{"(", SEPARATOR},

{")", SEPARATOR},

{"{", SEPARATOR},

{"}", SEPARATOR},

{",", SEPARATOR},

{";", SEPARATOR},

};

unordered\_map<string, Token> bool\_map = {

{"true",BOOL},

{"false",BOOL},

};

Token get\_keyword(const string& str); //判断是否是关键字

Token get\_operator(const string& str); //判断是否是运算符

Token get\_separator(const string& str); //判断是否是分隔符

Token get\_number(const string& str); //判断是否是整数常量

Token get\_bool(const string& str); //判断是否是字符常量

Token get\_explanatorynote(const string& str); //判断是否是注释

void write\_token(Token token, const string& word);

string in;

string out;

ifstream input\_file;

ofstream output\_file;

string find\_operator(const string& str, int i);

string find\_bool(const string& str, int i);

string find\_separator(const string& str, int i);

int findNthCharLine(const std::string& filename, int n, string& result);

};

lexical\_analyzer::lexical\_analyzer(const string& input\_file\_name, const string& output\_file\_name) {

input\_file.open(input\_file\_name);

output\_file.open(output\_file\_name);

in = input\_file\_name;

out = input\_file\_name;

}

lexical\_analyzer::~lexical\_analyzer() {

input\_file.close();

output\_file.close();

}

Token lexical\_analyzer::get\_keyword(const string& str) {

auto it = keyword\_map.find(str);

if (it != keyword\_map.end()) {

return it->second;

}

return IDENTIFIER;

}

Token lexical\_analyzer::get\_bool(const string& str) {

auto it = bool\_map.find(str);

if (it != bool\_map.end()) {

return it->second;

}

return IDENTIFIER;

}

Token lexical\_analyzer::get\_operator(const string& str) {

auto it = operator\_map.find(str);

if (it != operator\_map.end()) {

return it->second;

}

return INVALID\_TOKEN;

}

Token lexical\_analyzer::get\_separator(const string& str) {

auto it = separator\_map.find(str);

if (it != separator\_map.end()) {

return it->second;

}

return INVALID\_TOKEN;

}

Token lexical\_analyzer::get\_number(const string& str) {

for (char c : str) {

if (!isdigit(c)) {

return INVALID\_TOKEN;

}

}

return NUMBER;

}

void lexical\_analyzer::write\_token(Token token, const string& word) {

switch (token) {

/\*enum Token {

IDENTIFIER, //标识符

NUMBER, //整常数

BOOL, //布尔常量

KEYWORD, //关键字

OPERATOR, //运算符

SEPARATOR, //分隔符

INVALID\_TOKEN, //无效词

};\*/

case BOOL:

output\_file << "BL " << word << endl;

break;

case KEYWORD:

output\_file << "KW " << word << endl;

break;

case IDENTIFIER:

output\_file << "ID " << word << endl;

break;

case NUMBER:

output\_file << "NU " << word << endl;

break;

case OPERATOR:

output\_file << "OP " << word << endl;

break;

case SEPARATOR:

output\_file << "SP " << word << endl;

break;

default:

break;

}

}

void CreatNext\_E(string p, int next[]) {

int lenP = p.size();

int i = 0, j = -1;

next[0] = -1;

while (i < lenP - 1) {

if (j < 0 || p[i] == p[j]) {

i++;

j++;

next[i] = p[i] == p[j] ? next[j] : j; //此句代码进行了改进

}

else {

j = next[j];

}

}

}

int KMP\_find\_string\_in\_string(string t, string p, int current) {

int lenT = t.size();

int lenP = p.size();

int\* next = new int[lenP];

//CreatNext(p, next);

CreatNext\_E(p, next);

int i, j;

for (i = current; i <= lenT - lenP; ) {

for (j = 0; j < lenP; ++j) {

if (t[i + j] != p[j]) {

i += j - next[j];

break;

}

}

if (j == lenP) {

return i;

}

}

return -1;

}

int BF\_find\_string\_in\_string(string t, string p, int current) {

int lenT = t.size();

int lenP = p.size();

int i, j;

for (i = current; i <= lenT - lenP; ++i) {

for (j = 0; j < lenP; ++j) {

if (t[i + j] != p[j]) {

break;

}

}

if (j == lenP) {

return i;

}

}

return -1;

}

string lexical\_analyzer::find\_operator(const string& str, int i) {

int flag = 0;

unordered\_map<string, Token>::iterator iter;

for (iter = operator\_map.begin(); iter != operator\_map.end(); iter++)

{

flag = 0;

flag = KMP\_find\_string\_in\_string(str, iter->first, i);

if (flag == i) { return iter->first; }

}

return "";

}

string lexical\_analyzer::find\_bool(const string& str, int i) {

int flag = 0;

unordered\_map<string, Token>::iterator iter;

for (iter = bool\_map.begin(); iter != bool\_map.end(); iter++)

{

flag = 0;

flag = KMP\_find\_string\_in\_string(str, iter->first, i);

if (flag == i) { return iter->first; }

}

return "";

}

string lexical\_analyzer::find\_separator(const string& str, int i) {

int flag = 0;

unordered\_map<string, Token>::iterator iter;

for (iter = separator\_map.begin(); iter != separator\_map.end(); iter++)

{

flag = 0;

flag = KMP\_find\_string\_in\_string(str, iter->first, i);

if (flag == i) { return iter->first; }

}

return "";

}

std::vector<pair<string, string>> lexical\_analyzer::result\_to\_token(const string& ifn) {

vector<pair<string, string>> result;

ifstream input(ifn);

pair<string, string> current;

input >> current.first;

input >> current.second;

while (!input.eof()) {

result.push\_back(current);

input >> current.first;

input >> current.second;

}

return result;

}

int lexical\_analyzer::findNthCharLine(const std::string& filename, int n, string& result) {

using namespace std;

ifstream file(filename); // 打开文件

if (!file) {

cerr << "无法打开文件: " << filename << std::endl; // 如果无法打开文件，输出错误信息

return -1;

}

int lineNumber = 0;

int charCount = 0;

string line;

bool inMultilineComment = false; // 标记是否处于多行注释中

while (std::getline(file, line)) { // 逐行读取文件

charCount += line.size() + 1;

lineNumber++; // 行号递增

if (charCount > n)break;

}

if (charCount < n) {

std::cerr << "文件中字符数量少于 " << n << std::endl; // 如果文件中的字符串数量少于 n，输出错误信息

return -1;

}

return lineNumber;

}

bool lexical\_analyzer::analyze() {

string input\_string;

string string\_current;

//读入一整行

while (!input\_file.eof()) {

getline(input\_file, string\_current);

input\_string += string\_current + "\n";

}

string word;

Token token = INVALID\_TOKEN;

int i = 0;

while (i < input\_string.size()) {

char c = input\_string[i];

//如果当前运算符是空字符，忽略并继续

if (input\_string[i] < 0 || input\_string[i]>127) {

string w;

cout << "第" << findNthCharLine(in, i + 1, w) << "行，字符串\"" << word << "\"" << "后有一个非法字符串";

return false;

}

else if (isspace(c)) {

i++;

continue;

}

word.clear();

token = INVALID\_TOKEN;

//如果当前字符是'/'，判断是否是注释

if (i < input\_string.size() && input\_string[i] == '/' && input\_string[i + 1] == '/') {

while (input\_string[i] != '\n')i++;

i++;

continue;

}

if (i < input\_string.size() && input\_string[i] == '/' && input\_string[i + 1] == '\*') {

while (i < input\_string.size() && !(input\_string[i] == '\*' && input\_string[i + 1] == '/'))

i++;

i += 2;

continue;

}

if (find\_bool(input\_string, i) != "") {

string c = find\_bool(input\_string, i);

word += c;

i += c.length();

token = get\_bool(c);

}

//如果当前字符是字母

else if (isalpha(c)) {

//如果为字母或数字就全部读入

while (i < input\_string.size() && isalnum(input\_string[i])) {

word += input\_string[i];

i++;

}

token = get\_keyword(word);

if (token == IDENTIFIER) {

token = IDENTIFIER;

}

}

else if (isdigit(c)) {

while (i < input\_string.size() && isdigit(input\_string[i])) {

word += input\_string[i];

i++;

if (input\_string[i] < 0 || input\_string[i]>127) {

string a;

cout << "第" << findNthCharLine(in, i, a) << "行" << "的字串" << word << endl << "后有一个非法字符,请检查相关语法";

return false;

}

}

token = get\_number(word);

}

else if (find\_operator(input\_string, i) != "") {

string c = find\_operator(input\_string, i);

word += c;

i += c.length();

token = get\_operator(c);

}

else if (find\_separator(input\_string, i) != "") {

string c = find\_separator(input\_string, i);

word += c;

i += c.length();

token = get\_separator(c);

}

else {

word += input\_string[i];

string a;

cout << "第" << findNthCharLine(in, i, a) << "行" << "的字符: " << word << endl << "是一个非法字符,请检查相关语法";

return false;

// 无效字符

}

if (token != INVALID\_TOKEN) {

write\_token(token, word);

}

}

return true;

}

#pragma once

/\*经过分析

FIRST集：

FIRST(PROG) = { {,}

FIRST(DECLS) = { int, bool, ε }

FIRST(DECLS') = { int, bool, ε }

FIRST(DECL) = { int, bool }

FIRST(NAMES) = { id }

FIRST(NAMES') = { ,, ε }

FIRST(NAME) = { id }

FIRST(STMTS) = { id, if, while, {, read, write, ε }

FIRST(STMTS') = { id, if, while, {, read, write, ε }

FIRST(STMT) = { id, if, while, {, read, write }

FIRST(ELSESTMT) = { else, ε }

FIRST(EXPR) = { id, number, (, -, }

FIRST(ADD') = { +, -, ε }

FIRST(ADD) = { +, - }

FIRST(TERM) = { id, number, (, -, }

FIRST(MUL') = { \*, /, ε }

FIRST(MUL) = { \*, / }

FIRST(NEGA) = { id, number, (, -, }

FIRST(FACTOR) = { id, number, ( }

FIRST(BOOL) = { id, number, (, !, -, }

FIRST(BOOL') = { ||, ε }

FIRST(JOIN) = { id, number, (, !, -, }

FIRST(JOIN') = { &&, ε }

FIRST(NOT) = { id, number, (, !, -, }

FIRST(REL) = { id, number, (, -, }

FIRST(ROP) = { >, >=, <, <=, ==, != }

FOLLOW集：

FOLLOW(PROG) = {$}

FOLLOW(DECLS) = FOLLOW(DECLS') = {id, if, while, {, read, write, $}

FOLLOW(DECL) = {int, bool, id, if, while, {, read, write, $}

FOLLOW(NAMES) = FOLLOW(NAMES') = {;}

FOLLOW(NAME) = {,, ;}

FOLLOW(STMTS) = FOLLOW(STMTS') = {id, if, while, {, read, write, }, $}

FOLLOW(STMT) = {id, if, while, {, read, write, }, $}

FOLLOW(ELSESTMT) = {id, if, while, {, read, write, }, $}

FOLLOW(EXPR) = {), ;, >, >=, <, <=, ==, !=}

FOLLOW(ADD') = FOLLOW(ADD) = {), ;, >, >=, <, <=, ==, !=}

FOLLOW(TERM) = {+, -, ), ;, >, >=, <, <=, ==, !=}

FOLLOW(MUL') = FOLLOW(MUL) = {+, -, ), ;, >, >=, <, <=, ==, !=}

FOLLOW(NEGA) = {\*, /, +, -, ), ;, >, >=, <, <=, ==, !=}

FOLLOW(FACTOR) = {\*, /, +, -, ), ;, >, >=, <, <=, ==, !=}

FOLLOW(BOOL) = FOLLOW(BOOL') = {;, }

FOLLOW(JOIN) = FOLLOW(JOIN') = {||, ;, }

FOLLOW(NOT) = {&&, ||, ;, }

FOLLOW(REL) = {&&, ||, ;, }

FOLLOW(ROP) = {id, number, (, -, }

SELECT集:

SELECT(PROG → { DECLS STMTS }) = { { }

SELECT(DECLS → DECL DECLS') = { int, bool }

SELECT(DECLS → empty) = { id, if, while, {, read, write, $ }

SELECT(DECLS' → DECL DECLS') = { int, bool }

SELECT(DECLS' → empty) = { id, if, while, {, read, write, $ }

SELECT(DECL → int NAMES ;) = { int }

SELECT(DECL → bool NAMES ;) = { bool }

SELECT(NAMES → NAME NAMES') = { id }

SELECT(NAMES' → , NAME NAMES') = { , }

SELECT(NAMES' → empty) = { ; }

SELECT(NAME → id) = { id }

SELECT(STMTS → STMT STMTS') = { id, if, while, {, read, write }

SELECT(STMTS → empty) = { }, $ }

SELECT(STMTS' → STMT STMTS') = { id, if, while, {, read, write }

SELECT(STMTS' → empty) = { }, $ }

SELECT(STMT → id STMT')={id}

SELECT(STMT → = EXPR;)={=}

SELECT(STMT → := BOOL;)={:=}

SELECT(STMT → if id then STMT ELSESTMT) = { if }

SELECT(ELSESTMT → else STMT) = { else }

SELECT(ELSESTMT → empty) = { id, if, while, {, read, write, }, $ }

SELECT(STMT → while id do STMT) = { while }

SELECT(STMT → { STMTS }) = { { }

SELECT(STMT → read id ;) = { read }

SELECT(STMT → write id ;) = { write }

SELECT(EXPR → TERM ADD') = { id, number, (, - }

SELECT(ADD' → ADD TERM ADD') = { +, - }

SELECT(ADD' → empty) = { ), ;, >, >=, <, <=, ==, != }

SELECT(ADD → +) = { + }

SELECT(ADD → -) = { - }

SELECT(TERM → NEGA MUL') = { id, number, (, - }

SELECT(MUL' → MUL NEGA MUL') = { \*, / }

SELECT(MUL' → empty) = { +, -, ), ;, >, >=, <, <=, ==, != }

SELECT(MUL → \*) = { \* }

SELECT(MUL → /) = { / }

SELECT(NEGA → FACTOR) = { id, number, ( }

SELECT(NEGA → - FACTOR) = { - }

SELECT(FACTOR → ( EXPR )) = { ( }

SELECT(FACTOR → id) = { id }

SELECT(FACTOR → number) = { number }

SELECT(BOOL → JOIN BOOL') = { id, number, (, !, - }

SELECT(BOOL' → || JOIN BOOL') = { || }

SELECT(BOOL' → empty) = { ;, } }

SELECT(JOIN → NOT JOIN') = { id, number, (, !, - }

SELECT(JOIN' → && NOT JOIN') = { && }

SELECT(JOIN' → empty) = { ;, ||, } }

SELECT(NOT → REL) = { id, number, (, - }

SELECT(NOT → ! REL) = { ! }

SELECT(REL → EXPR ROP EXPR) = { id, number, (, - }

SELECT(ROP → >) = { > }

SELECT(ROP → >=) = { >= }

SELECT(ROP → <) = { < }

SELECT(ROP → <=) = { <= }

SELECT(ROP → ==) = { == }

SELECT(ROP → !=) = { != }

\*/

/\*以下是我的代码\*/

#include "lexical\_analyzer.h"

#include <iostream>

#include <fstream>

#include <string>

#include<sstream>

#include <vector>

#include<stack>

using namespace std;

struct TreeNode {

pair<string, string> val;

TreeNode\* N1;

TreeNode\* N2;

TreeNode\* N3;

TreeNode\* N4;

TreeNode\* N5;

TreeNode\* N6;

TreeNode(pair<string, string> x) : val(x), N1(nullptr), N2(nullptr), N3(nullptr), N4(nullptr), N5(nullptr), N6(nullptr) {}

};

//语法树模块及相关操作

class tree {

public:

tree();

~tree();

TreeNode\* getroot() {

return root;

}

bool add\_node(TreeNode\* current, pair<string, string> data, TreeNode\*& NowInsert);//在current下面插入一个子节点。从左向右的顺序，如果满了，返回false.

string serialize(TreeNode\* root); // 树转化为字符串

TreeNode\* NewNode(pair<string, string>); //创建一个新的树节点

TreeNode\* deserialize(string data); // 字符串转化为树

bool isEndNode(TreeNode\* cur);

private:

TreeNode\* root;

TreeNode\* deserializeHelper(const vector<pair<string, string>>& vals, int& pos);//字符串转化为树的辅助函数

};

bool tree::isEndNode(TreeNode\* cur) {

if (cur->N1 == NULL)return true;

else

return false;

}

tree::tree() {

root = nullptr;

}

tree::~tree() {

root = nullptr;

}

bool tree::add\_node(TreeNode\* current, pair<string, string> data, TreeNode\*& NowInsert) {

if (root == nullptr) {

root = NewNode(data);

return true;

}

if (current->N1 == nullptr) {

current->N1 = NewNode(data);

NowInsert = current->N1;

}

else if (current->N2 == nullptr) {

current->N2 = NewNode(data);

NowInsert = current->N2;

}

else if (current->N3 == nullptr) {

current->N3 = NewNode(data);

NowInsert = current->N3;

}

else if (current->N4 == nullptr) {

current->N4 = NewNode(data);

NowInsert = current->N4;

}

else if (current->N5 == nullptr) {

current->N5 = NewNode(data);

NowInsert = current->N5;

}

else if (current->N6 == nullptr) {

current->N6 = NewNode(data);

NowInsert = current->N6;

}

else

return false;

return true;

}

string tree::serialize(TreeNode\* root) {

if (root == nullptr) return "";

stack<TreeNode\*> s;

s.push(root);

string res = "";

while (!s.empty()) {

TreeNode\* cur = s.top();

s.pop();

if (cur != nullptr) {

res += (cur->val.first) + " ";

res += (cur->val.second) + " ";

s.push(cur->N6);

s.push(cur->N5);

s.push(cur->N4);

s.push(cur->N3);

s.push(cur->N2);

s.push(cur->N1);

}

else {

res += "#";

res += " ";

}

}

return res;

}

TreeNode\* tree::deserialize(string data) {

stringstream ss(data);

pair<string, string> current\_pair;

vector<pair<string, string>> val;

string current;

while (ss >> current) {

current\_pair.first = current;

if (current != "#")

ss >> current;

current\_pair.second = current;

val.push\_back(current\_pair);

}

int pos = 0;

return deserializeHelper(val, pos);

}

TreeNode\* tree::deserializeHelper(const vector<pair<string, string>>& vals, int& pos) {

if (vals[pos].first == "#") {

pos++;

return nullptr;

}

TreeNode\* node = new TreeNode(vals[pos++]);

node->N1 = deserializeHelper(vals, pos);

node->N2 = deserializeHelper(vals, pos);

node->N3 = deserializeHelper(vals, pos);

node->N4 = deserializeHelper(vals, pos);

node->N5 = deserializeHelper(vals, pos);

node->N6 = deserializeHelper(vals, pos);

return node;

}

TreeNode\* tree::NewNode(pair<string, string> data) {

TreeNode\* node = new TreeNode(data);

return node;

}

// 语法分析模块

class Parser {

public:

Parser(const string& InputFile, const string& middleFile, const string& OutputFile);

~Parser();

void print(); //输出预测分析表

TreeNode\* deserialize(const string& input); //序列化结果文件的反序列化，反序列化是否成功

void serialize(const string& output, TreeNode\* root); //序列化结果文件的反序列化结果文件的序列化，通过与序列化结果文件对比来验证序列化和反序列化函数

bool do\_parser(const vector<pair<string, string>>& input, TreeNode\*& result); //语法分析函数

bool do\_and\_write(); //词法分析、语法分析及所有结果输出函数

private:

bool isComment(const std::string& line, size\_t& pos); //当前行是否存在注释

bool is\_end\_character(const string& a); //当前字符串是否是终结符

string in; //输入文件名

string out; //语法分析结果输出文件名

string middle; //词法分析结果输出文件名

int findNthStringLine(const std::string& filename, int n, string& result); //找到当前文件中第i个字符串的函数。方便语法分析错误处理

vector<vector<pair<vector<string>, vector<string>>>>LL1\_t; //预测分析表。最外层数组表示不同的非终结符对应的每个状态。第二层数组表示每个状态的多个语句。第三层数组，表示每个状态的产生式右部非终结符集合和相应select集中终结符结合

void init\_LL1\_able();

bool do\_parser\_input\_change(vector<pair<string, string>>& result, const vector<pair<string, string>>& input); //处理词法分析结果，使其更适合进行语法分析

int find\_status(string a); //在预测分析表中找到相应的非终结符

string true\_string(const string& a); //当前字符串如果分隔符算一个字符串，则当前字符串的所有分隔符用空格隔开

int judge\_status(string a, int status); //在预测分析表中，找到相应的产生式

};

TreeNode\* Parser::deserialize(const string& input) {

ifstream in(input);

TreeNode\* result;

string current;

string middle;

middle.clear();

while (in >> current) {

middle += current + " ";

}

middle += '\0';

tree t;

result = t.deserialize(middle);

return result;

}

void Parser::serialize(const string& output, TreeNode\* root) {

ofstream out(output);

tree t;

string middle;

middle = t.serialize(root);

out << middle;

}

bool Parser::isComment(const std::string& line, size\_t& pos) {

pos = line.find("//");

if (pos != std::string::npos) {

return true;

}

return false;

}

string Parser::true\_string(const string& a) {

string result;

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

if (a[i] == '(' || a[i] == ')' || a[i] == '{' || a[i] == '}' || a[i] == ',' || a[i] == ';') {

result += " ";

result += a[i];

result += " ";

}

else {

result += a[i];

}

}

return result;

}

int Parser::findNthStringLine(const std::string& filename, int n, string& result) {

using namespace std;

ifstream file(filename); // 打开文件

if (!file) {

cerr << "无法打开文件: " << filename << std::endl; // 如果无法打开文件，输出错误信息

return -1;

}

int lineNumber = 0;

int stringCount = 0;

string line;

bool inMultilineComment = false; // 标记是否处于多行注释中

while (std::getline(file, line)) { // 逐行读取文件

lineNumber++; // 行号递增

size\_t commentPos;

if (isComment(line, commentPos)) { // 判断当前行是否为单行注释

line = line.substr(0, commentPos); // 删除注释部分

}

size\_t multiCommentStart = line.find("/\*"); // 查找多行注释起始符

size\_t multiCommentEnd = line.find("\*/"); // 查找多行注释结束符

if (inMultilineComment) { // 如果处于多行注释中

if (multiCommentEnd != std::string::npos) { // 如果找到多行注释结束符

inMultilineComment = false; // 设置多行注释标记为 false

line = line.substr(multiCommentEnd + 2); // 删除注释部分

}

else {

continue; // 继续读取下一行

}

}

else {

if (multiCommentStart != std::string::npos) { // 如果找到多行注释起始符

inMultilineComment = true; // 设置多行注释标记为 true

line = line.substr(0, multiCommentStart); // 删除注释部分

}

}

if (line.empty()) { // 如果当前行为空，跳过处理

continue;

}

line = true\_string(line);

std::istringstream iss(line); // 创建字符串流以处理当前行

std::string word;

bool is\_first = true;

while (iss >> word) { // 读取当前行的每个字符串

stringCount++; // 字符串计数递增

if (stringCount == n - 1)result = word;

if (stringCount == n && is\_first) { // 如果达到目标字符串数量

return lineNumber - 1; // 返回行号

}

if (stringCount == n && !is\_first) { // 如果达到目标字符串数量

return lineNumber; // 返回行号

}

if (is\_first)

is\_first = false;

}

}

std::cerr << "文件中字符串数量少于 " << n << std::endl; // 如果文件中的字符串数量少于 n，输出错误信息

return -1;

}

bool Parser::is\_end\_character(const string& a) {

if (a == "int" || a == "bool" || a == "if" || a == "then" || a == "while" || a == "do" || a == "read" || a == "write" || a == "else" || a == "+" || a == "-" || a == "\*" || a == "/" || a == "||" || a == "&&" || a == "!" || a == ">=" || a == ">" || a == "<=" || a == "<" || a == ":=" || a == "==" || a == "=" || a == "(" || a == ")" || a == "{" || a == "}" || a == ";" || a == "," || a == "number" || a == "id")

return true;

return false;

}

std::string readFile(const std::string& fileName) {

std::ifstream file(fileName);

std::string content((std::istreambuf\_iterator<char>(file)), std::istreambuf\_iterator<char>());

return content;

}

Parser::Parser(const string& InputFile, const string& middleFile, const string& OutputFile) {

in = InputFile;

out = OutputFile;

middle = middleFile;

pair<vector<string>, vector<string>> current;

vector<pair<vector<string>, vector<string>>> current1;

//PROG

current.first = { "{","DECLS","STMTS","}" };

current.second = { "{" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//DECLS

current.first = { "DECL","DECLS'" };

current.second = { "int", "bool" };

current1.push\_back(current);

current.first = { "empty" };

current.second = { "id", "if", "while", "{", "read", "write", "$" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//DELCS'

current.first = { "DECL","DECLS'" };

current.second = { "int", "bool" };

current1.push\_back(current);

current.first = { "empty" };

current.second = { "id", "if", "while", "{", "read", "write", "$" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//DECL

current.first = { "int","NAMES",";" };

current.second = { "int" };

current1.push\_back(current);

current.first = { "bool","NAMES",";" };

current.second = { "bool" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//NAMES

current.first = { "NAME","NAMES'" };

current.second = { "id" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//NAMES'

current.first = { ",","NAME","NAMES'" };

current.second = { "," };

current1.push\_back(current);

current.first = { "empty" };

current.second = { ";" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//NAME

current.first = { "id" };

current.second = { "id" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//STMTS

current.first = { "STMT","STMTS'" };

current.second = { "id","if","while", "{", "read", "write" };

current1.push\_back(current);

current.first = { "empty" };

current.second = { "}", "$" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//STMTS'

current.first = { "STMT","STMTS'" };

current.second = { "id","if","while", "{", "read", "write" };

current1.push\_back(current);

current.first = { "empty" };

current.second = { "}", "$" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//STMT

current.first = { "id","STMT'" };

current.second = { "id" };

current1.push\_back(current);

current.first = { "if","id","then","STMT","ELSESTMT" };

current.second = { "if" };

current1.push\_back(current);

current.first = { "while","id","do","STMT" };

current.second = { "while" };

current1.push\_back(current);

current.first = { "{","STMTS","}" };

current.second = { "{" };

current1.push\_back(current);

current.first = { "read","id",";" };

current.second = { "read" };

current1.push\_back(current);

current.first = { "write","id",";" };

current.second = { "write" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//STMT'

current.first = { "=","EXPR",";" };

current.second = { "=" };

current1.push\_back(current);

current.first = { ":=","BOOL",";" };

current.second = { ":=" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//ELSESTMT

current.first = { "else","STMT" };

current.second = { "else" };

current1.push\_back(current);

current.first = { "empty" };

current.second = { "id","if","while","{","read","write","}","$" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//EXPR

current.first = { "TERM","ADD'" };

current.second = { "id","number","(","-" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//ADD'

current.first = { "ADD","TERM","ADD'" };

current.second = { "+","-" };

current1.push\_back(current);

current.first = { "empty" };

current.second = { ")",";",">",">=","<","<=","==","!=" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//ADD

current.first = { "+" };

current.second = { "+" };

current1.push\_back(current);

current.first = { "-" };

current.second = { "-" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//TERM

current.first = { "NEGA","MUL'" };

current.second = { "id","number","(","-" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//MUL'

current.first = { "MUL","NEGA","MUL'" };

current.second = { "\*","/" };

current1.push\_back(current);

current.first = { "empty" };

current.second = { "+","-",")",";",">",">=","<","<=" ,"==" ,"!=" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//MUL

current.first = { "\*" };

current.second = { "\*" };

current1.push\_back(current);

current.first = { "/" };

current.second = { "/" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//NEGA

current.first = { "FACTOR" };

current.second = { "id","number", "(" };

current1.push\_back(current);

current.first = { "-","FACTOR" };

current.second = { "-" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//FACTOR

current.first = { "(","EXPR",")" };

current.second = { "(" };

current1.push\_back(current);

current.first = { "id" };

current.second = { "id" };

current1.push\_back(current);

current.first = { "number" };

current.second = { "number" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//BOOL

current.first = { "JOIN","BOOL'" };

current.second = { "id","number", "(", "!", "-" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//BOOL'

current.first = { "||","JOIN","BOOL'" };

current.second = { "||" };

current1.push\_back(current);

current.first = { "empty" };

current.second = { ";","}" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//JOIN

current.first = { "NOT","JOIN'" };

current.second = { "id","number", "(", "!", "-" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//JOIN'

current.first = { "&&","NOT","JOIN'" };

current.second = { "&&" };

current1.push\_back(current);

current.first = { "empty" };

current.second = { ";", "||" ,"}" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//NOT

current.first = { "REL" };

current.second = { "id","number","(","-" };

current1.push\_back(current);

current.first = { "!","REL" };

current.second = { "!" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//REL

current.first = { "EXPR","ROP","EXPR" };

current.second = { "id","number","(","-" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

//ROP

current.first = { ">" };

current.second = { ">" };

current1.push\_back(current);

current.first = { ">=" };

current.second = { ">=" };

current1.push\_back(current);

current.first = { "<" };

current.second = { "<" };

current1.push\_back(current);

current.first = { "<=" };

current.second = { "<=" };

current1.push\_back(current);

current.first = { "==" };

current.second = { "==" };

current1.push\_back(current);

current.first = { "!=" };

current.second = { "!=" };

current1.push\_back(current);

LL1\_t.push\_back(current1);

current1.clear();

}

int status\_find(const string a) {

if (a == "PROG")return 1;

else if (a == "DECLS")return 2;

else if (a == "DECLS'")return 3;

else if (a == "DECL")return 4;

else if (a == "NAMES")return 5;

else if (a == "NAMES'")return 6;

else if (a == "NAME")return 7;

else if (a == "STMTS")return 8;

else if (a == "STMTS'")return 9;

else if (a == "STMT")return 10;

else if (a == "STMT'")return 11;

else if (a == "ELSESTMT")return 12;

else if (a == "EXPR")return 13;

else if (a == "ADD'")return 14;

else if (a == "ADD")return 15;

else if (a == "TERM")return 16;

else if (a == "MUL'")return 17;

else if (a == "MUL")return 18;

else if (a == "NEGA")return 19;

else if (a == "FACTOR")return 20;

else if (a == "BOOL")return 21;

else if (a == "BOOL'")return 22;

else if (a == "JOIN")return 23;

else if (a == "JOIN'")return 24;

else if (a == "NOT")return 25;

else if (a == "REL")return 26;

else if (a == "ROP")return 27;

else return -1;

}

Parser::~Parser() {

}

void Parser::print() {

for (int i = 0; i < LL1\_t.size(); i++) {

cout << i + 1 << endl;

for (int j = 0; j < LL1\_t[i].size(); j++) {

for (int m = 0; m < LL1\_t[i][j].first.size(); m++) {

cout << LL1\_t[i][j].first[m] << " ";

}

cout << "--------------------";

for (int m = 0; m < LL1\_t[i][j].second.size(); m++) {

cout << LL1\_t[i][j].second[m] << " ";

}

cout << endl;

}

}

}

int Parser::find\_status(string a) {

if (a == "PROG")return 1;

else if (a == "DECLS")return 2;

else if (a == "DECLS'")return 3;

else if (a == "DECL")return 4;

else if (a == "NAMES")return 5;

else if (a == "NAMES'")return 6;

else if (a == "NAME")return 7;

else if (a == "STMTS")return 8;

else if (a == "STMTS'")return 9;

else if (a == "STMT")return 10;

else if (a == "STMT'")return 11;

else if (a == "ELSESTMT")return 12;

else if (a == "EXPR")return 13;

else if (a == "ADD'")return 14;

else if (a == "ADD")return 15;

else if (a == "TERM")return 16;

else if (a == "MUL'")return 17;

else if (a == "MUL")return 18;

else if (a == "NEGA")return 19;

else if (a == "FACTOR")return 20;

else if (a == "BOOL")return 21;

else if (a == "BOOL'")return 22;

else if (a == "JOIN")return 23;

else if (a == "JOIN'")return 24;

else if (a == "NOT")return 25;

else if (a == "REL")return 26;

else if (a == "ROP")return 27;

else return -1;

}

int Parser::judge\_status(string a, int status) {

for (int i = 0; i < LL1\_t[status - 1].size(); i++) {

for (int m = 0; m < LL1\_t[status - 1][i].second.size(); m++) {

if (LL1\_t[status - 1][i].second[m] == a)return i;

}

}

return -1;

}

bool Parser::do\_parser\_input\_change(vector<pair<string, string>>& result, const vector<pair<string, string>>& input) {

result.clear();

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

if (input[i].first == "BL")

result.push\_back({ "bool",input[i].second });

else if (input[i].first == "KW")

result.push\_back({ input[i].second,input[i].second });

else if (input[i].first == "ID")

result.push\_back({ "id",input[i].second });

else if (input[i].first == "NU")

result.push\_back({ "number",input[i].second });

else if (input[i].first == "OP")

result.push\_back({ input[i].second,input[i].second });

else if (input[i].first == "SP")

result.push\_back({ input[i].second,input[i].second });

else {

return false;

}

}

result.push\_back({ "$","$" });

return true;

}

bool Parser::do\_parser(const vector<pair<string, string>>& input, TreeNode\*& result) {

tree t;

stack<TreeNode\*> stack\_b;

TreeNode\* tree\_node\_current = (TreeNode\*)malloc(sizeof(TreeNode));

TreeNode\* tree\_node\_current1 = (TreeNode\*)malloc(sizeof(TreeNode));

vector<pair<string, string>> true\_input;

if (!do\_parser\_input\_change(true\_input, input))return false; //对词法分析器结果进行处理

int judge\_seperator[2] = { 0,0 };

if (true\_input[0].first != "{") {

string error\_string;

int current\_line = findNthStringLine(in, 1, error\_string);

cout << "第" << current\_line << "行: " << endl;

cout << "起始符号错误！代码应该以 \"{\" 为起始符号，你的起始符号为: " << true\_input[0].first << endl;

return false;

}

for (int i = 0; i < true\_input.size(); i++) {

if (true\_input[i].first == "{")judge\_seperator[0]++;

else if (true\_input[i].second == "}")judge\_seperator[0]--;

else if (true\_input[i].second == "(")judge\_seperator[1]++;

else if (true\_input[i].second == ")")judge\_seperator[1]--;

}

if (judge\_seperator[0] != 0 || judge\_seperator[1] != 0) {

if (judge\_seperator[0] != 0)

cout << "{}未得到匹配，请仔细检查你的语法" << endl;

if (judge\_seperator[1] != 0)

cout << "()未得到匹配，请仔细检查你的语法" << endl;

return false;

}

stack<TreeNode\*> a; //语法分析程序的栈

string current; //当前语法分析栈的栈顶元素

int flag = 0; //词法分析结果的指针

t.add\_node(t.getroot(), { "PROG","PROG" }, tree\_node\_current1); //创建根节点

TreeNode\* end\_note = new TreeNode({ "$","$" });

a.push(end\_note);

a.push(t.getroot()); //根节点入栈

int status; //当前字符串对应状态

int number; //当前字符串对应状态下的第几语句

while (flag < true\_input.size()) {

tree\_node\_current = a.top(); //取出栈顶元素

a.pop();

current = tree\_node\_current->val.first;

if (current == "empty") {

t.add\_node(tree\_node\_current, tree\_node\_current->val, tree\_node\_current1);

continue;

}

else if (current == true\_input[flag].first) { //如果栈顶元素是终结符且符合，栈顶元素出栈，分析向后移动一位

t.add\_node(tree\_node\_current, { true\_input[flag].first ,true\_input[flag].second }, tree\_node\_current1);

flag++;

continue;

}

else if (current == "$" && true\_input[flag].first != "$") {

string error\_string;

int current\_line = findNthStringLine(in, flag + 1, error\_string);

cout << "第" << current\_line << "行: " << endl;

cout << "代码末尾多余非法字符：" << endl;

while (flag < true\_input.size() - 1)cout << true\_input[flag++].second << " ";

cout << endl;

return false;

}

else if (is\_end\_character(current)) { //终结符不匹配

string error\_string;

int current\_line = findNthStringLine(in, flag + 1, error\_string);

cout << "第" << current\_line << "行: " << endl;

cout << "字符串: " << error\_string << endl << "是一个无法匹配的字符串，请检查你的语法" << endl;

return false;

}

status = find\_status(current);

number = judge\_status(true\_input[flag].first, status);

if (number == -1) {

string error\_string;

int current\_line = findNthStringLine(in, flag + 1, error\_string);

cout << "第" << current\_line << "行: " << endl;

cout << "字符串: " << error\_string << endl << "是一个无法匹配的字符串，请检查你的语法" << endl;

return false;

}

for (int i = 0; i < LL1\_t[status - 1][number].first.size(); i++) {

if (!t.add\_node(tree\_node\_current, { LL1\_t[status - 1][number].first[i],LL1\_t[status - 1][number].first[i] }, tree\_node\_current1))return false; //分析入栈

stack\_b.push(tree\_node\_current1);

}

while (!stack\_b.empty()) {

a.push(stack\_b.top());

stack\_b.pop();

}

}

result = t.getroot();

return true;

}

// 将结果写入文件

void writeFile(const std::string& fileName, const std::string& content) {

std::ofstream file(fileName);

file << content;

}

bool Parser::do\_and\_write() {

ifstream input\_string(in);

if (!input\_string.is\_open()) {

ofstream out(in);

string path = \_\_FILE\_\_;

string now\_path = path.substr(0, path.size() - 6);

cout << "未设置源代码文件，请在目录: " << now\_path << "下，设置test.txt文件为非空!!!,您的源代码不能为空!!!" << endl;

input\_string.close();

return false;

}

char c;

input\_string >> c;

if (input\_string.eof()) {

string path = \_\_FILE\_\_;

string now\_path = path.substr(0, path.size() - 6);

cout << "未设置源代码文件，请在目录: " << now\_path << "下，设置test.txt文件为非空!!!,您的源代码不能为空!!!" << endl;

input\_string.close();

return false;

}

input\_string.close();

lexical\_analyzer a(in, middle);

if (a.analyze()) {

tree t;

TreeNode\* result = nullptr;

vector<pair<string, string>> in\_put;

in\_put = a.result\_to\_token(middle);

if (do\_parser(in\_put, result)) {

ofstream out\_\_put(out);

string result\_string = t.serialize(result);

out\_\_put << result\_string;

out\_\_put.close();

cout << "语法正确" << endl;

}

return true;

}

else

return false;

}

/\*

int、bool、if、then、while、do、read、write、else、+、-、\*、/、||、&&、!、>=、>、<=、<、:=、==、=、(、)、{、}、;、,、number、id

\*/

bool file\_same(const string& file1, const string& file2) {

ifstream input1(file1);

ifstream input2(file2);

string current1;

string current2;

bool result = true;

while ((input1 >> current1) && (input2 >> current2))

if (current1 != current2)

result = false;

return result;

}

#pragma once

#include"Parser.h"

#include <iostream>

#include <string>

#include <fstream>

#include<vector>

#include<utility>

using namespace std;

ofstream outputFile;

ifstream inputFile;

void processPROG(TreeNode\* node);

void processPROG(TreeNode\* node);

void processDECLS(TreeNode\* node);

void processDECLS\_(TreeNode\* node);

void processDECL(TreeNode\* node);

vector<string> processNAMES(TreeNode\* node);

vector<string> processNAMES\_(TreeNode\* node);

void processSTMTS(TreeNode\* node);

void processSTMTS\_(TreeNode\* node);

void processSTMT(TreeNode\* node);

void processSTMT\_(TreeNode\* root, string varName);

void processIfStmt(TreeNode\* node);

void processELSESTMT(TreeNode\* node);

void processWhileStmt(TreeNode\* node);

void processReadStmt(TreeNode\* node);

void processWriteStmt(TreeNode\* node);

void processEXPR(TreeNode\* node);

void processADD\_(TreeNode\* node);

void processTERM(TreeNode\* node);

void processMUL\_(TreeNode\* node);

void processNEGA(TreeNode\* node);

void processFACTOR(TreeNode\* node);

void processBOOL(TreeNode\* node);

void processBOOL\_(TreeNode\* node);

void processJOIN(TreeNode\* node);

void processJOIN\_(TreeNode\* node);

void processNOT(TreeNode\* node);

void processREL(TreeNode\* node);

//定义符号表

vector<pair<pair<string,string>,int>>symboltable;

int label\_id = 0;

int getlabel\_id() {

return label\_id++;

}

void emit(string instruction) {

outputFile << "\t" << instruction << endl;

}

void int\_stack\_init() {

emit("#int栈初始化,t8为int栈指针");

emit("la $t8, int\_stack");

emit("addiu $t8, $t8, 508");

}

//t1入栈

void int\_stack\_push() {

emit("#t1入栈");

emit("addiu $t8, $t8, -4");

emit("sw $t1, 0($t8)");

}

//int出栈

void int\_stack\_pop\_1() {

emit("#出栈到t1");

emit("lw $t1, 0($t8)");

emit("addiu $t8, $t8, 4");

}

void int\_stack\_pop\_1\_2() {

emit("#出栈到t1与t2");

emit("lw $t1, 0($t8)");

emit("addiu $t8, $t8, 4");

emit("lw $t2, 0($t8)");

emit("addiu $t8, $t8, 4");

}

void bool\_stack\_init() {

emit("#bool栈初始化");

emit("la $t9, bool\_stack");

emit("addiu $t9, $t9, 127");

}

void bool\_stack\_push\_1() {

emit("#t1入栈");

emit("addiu $t9, $t9, -1");

emit("sb $t1, 0($t9)");

}

void bool\_stack\_pop\_1() {

emit("#出栈到t1");

emit("lb $t1, 0($t9)");

emit("addiu $t9, $t9, 1");

}

void bool\_stack\_pop\_1\_2() {

emit("#出栈到t1和t2");

emit("lb $t1, 0($t9)");

emit("addiu $t9, $t9, 1");

emit("lb $t2, 0($t9)");

emit("addiu $t9, $t9, 1");

}

void processPROG(TreeNode\* node) {

outputFile.open("final\_result.txt");

if (node->val.first == "PROG") {

// 输出汇编代码的开头部分

emit(".data");

emit("input\_stack:.space 1024");

emit("input\_buffer: .word 0");

emit("int\_stack: .word 128");

emit("bool\_stack: .byte 128 ");

emit(".text\n");

emit(".globl main\n");

emit("main:");

//初始化栈

emit("#初始化栈");

int\_stack\_init();

bool\_stack\_init();

// 处理DECLS和STMTS节点

if (node->N2 != nullptr) processDECLS(node->N2);

if (node->N3 != nullptr) processSTMTS(node->N3);

// 输出汇编代码的结尾部分

emit("li $v0, 10\n");

emit("syscall\n");

}

outputFile.close();

inputFile.open("final\_result.txt");

std::vector<std::string> lines;

std::string line;

while (std::getline(inputFile, line)) {

lines.push\_back(line);

}

inputFile.close();

outputFile.open("final\_result.txt");

for (int i = 0; i < 5; i++) {

emit(lines[i]);

}

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

if (symboltable[i].first.second == "int")

emit("\t"+symboltable[i].first.first + ": .word 0");

else

emit("\t"+symboltable[i].first.first + ": .byte 0");

}

for (int i = 5; i < lines.size(); i++) {

emit(lines[i]);

}

outputFile.close();

}

void processDECLS(TreeNode \* node) {

if (node == nullptr) {

return; // 如果节点为空，则直接返回

}

// 如果不为空，处理第一个DECL节点

processDECL(node->N1);

// 处理后续的DECLS'节点

processDECLS\_(node->N2);

}

void processDECLS\_(TreeNode \* node) {

if ((node == nullptr)||node->N1->val.first=="empty") {

return; // 如果节点为空，则直接返回

}

// 如果不为空，处理DECL节点

processDECL(node->N1);

// 处理后续的DECLS'节点

processDECLS\_(node->N2);

}

void processDECL(TreeNode\* node) {

if (node == nullptr) {

return; // 如果节点为空，则直接返回

}

// 从node的val获取类型

string type = node->N1->val.first;

// 处理NAMES节点，并返回变量名列表

vector<string> varNames = processNAMES(node->N2);

pair<string, string> cur;

// 对于每个变量名，添加到符号表中

for (const string& varName : varNames) {

// 在符号表中添加变量名和类型

cur.first = varName;

cur.second = type;

symboltable.push\_back(make\_pair(cur,-1));

}

}

vector<string> processNAMES(TreeNode\* node) {

vector<string> varNames;

if (node == nullptr) {

return varNames;

}

// 处理NAME节点

string varName = node->N1->N1->N1->val .second;

varNames.push\_back(varName);

// 递归处理NAMES'节点

vector<string> otherNames = processNAMES\_(node->N2);

varNames.insert(varNames.end(), otherNames.begin(), otherNames.end());

return varNames;

}

vector<string> processNAMES\_(TreeNode\* node) {

vector<string> varNames;

if (node == nullptr|| node->N1->val.first == "empty") {

return varNames;

}

// 处理NAME节点

string varName = node->N2->N1->N1->val.second;

varNames.push\_back(varName);

// 递归处理NAMES'节点

vector<string> otherNames = processNAMES\_(node->N3);

varNames.insert(varNames.end(), otherNames.begin(), otherNames.end());

return varNames;

}

void processSTMTS(TreeNode\* node) {

if (node == nullptr || node->N1->val.first == "empty") {

return;

}

// 处理STMT节点

processSTMT(node->N1);

// 递归处理STMTS'节点

processSTMTS\_(node->N2);

}

void processSTMTS\_(TreeNode\* node) {

if (node == nullptr || node->N1->val.first == "empty") {

return;

}

// 处理STMT节点

processSTMT(node->N1);

// 递归处理STMTS'节点

processSTMTS\_(node->N2);

}

void processSTMT(TreeNode\* node) {

if (node == nullptr) {

return;

}

if (node->N1->val.first == "id") { // 根据节点的子节点数量或特定标识来判断节点类型

// id STMT'

string varName = node->N1->N1->val.second; // 获取变量名

processSTMT\_(node->N2, varName);

}

else if (node->N1->val.first == "if") {

// if id then STMT ELSESTMT

processIfStmt(node);

}

else if (node->N1->val.first == "while") {

// while id do STMT

processWhileStmt(node);

}

else if (node->N1->val.first == "read") {

// read id ;

processReadStmt(node);

}

else if (node->N1->val.first == "write") {

// write id ;

processWriteStmt(node);

}

else if (node->N1->val.first == "{") {

// { STMTS }

processSTMTS(node->N2);

}

}

void processSTMT\_(TreeNode\* root, string varName) {

if (root->N1->val.first == "=") {

processEXPR(root->N2);

emit("# 将栈顶元素写回");

int\_stack\_pop\_1();

emit("la $t2, "+ varName);

emit("sw $t1, 0($t2)");

}

else if (root->N1->val.first == ":=") {

processBOOL(root->N2);

emit("# 将栈顶元素写回");

bool\_stack\_pop\_1();

emit("la $t2, " + varName);

emit("sb $t1, 0($t2)");

}

return;

}

void processIfStmt(TreeNode\* node) {

string label\_else = "else\_" + std::to\_string(label\_id); // 标记else语句块

string label\_end = "endif\_" + std::to\_string(label\_id); // 标记if-else语句结束

label\_id++; // 更新标签编号

// 将条件变量加载到$t0寄存器

string id\_type;

string id\_name = node->N2->N1->val.second;

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

if (symboltable[i].first.first == id\_name) {

id\_type = symboltable[i].first.second;

break;

}

}

if(id\_type=="int")

emit("lw $t0, " + id\_name);

else

emit("lb $t0, " + id\_name);

// 根据条件跳转到else语句块或者if语句块

emit("beqz $t0, " + label\_else); // 如果条件为假，则跳转到else语句块

processSTMT(node->N4); // 生成if语句块的代码

emit("j " + label\_end); // 无条件跳转到if-else语句结束

emit(label\_else + ":"); // 插入else语句块的标签

processELSESTMT(node->N5); // 如果存在else语句，则需要生成跳转到if语句块后面的代码

emit(label\_end + ":"); // 插入if-else语句结束的标签

}

void processELSESTMT(TreeNode\* node) {

if (node == nullptr || node->N1->val.first == "empty")

return;

else

processSTMT(node->N2);

}

void processWhileStmt(TreeNode\* node) {

std::string loopLabel = "loop" + std::to\_string(label\_id); // 循环起始位置的标签

std::string endLoopLabel = "endloop" + std::to\_string(label\_id); // 循环结束位置的标签

label\_id++; // 更新标签编号

// 生成循环条件部分的代码

string id\_type;

string id\_name = node->N2->N1->val.second;

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

if (symboltable[i].first.first == id\_name) {

id\_type = symboltable[i].first.second;

break;

}

}

emit(loopLabel + ":"); // 插入循环起始位置的标签

// 生成循环条件判断部分的代码

if (id\_type == "int")

emit("lw $t0, " + id\_name);

else

emit("lb $t0, " + id\_name);

emit("beqz $t0, " + endLoopLabel); // 如果条件为假，则跳出循环

// 生成循环体部分的代码

processSTMT(node->N4);

emit("j " + loopLabel); // 跳回循环起始位置

emit(endLoopLabel + ":"); // 插入循环结束位置的标签

}

void processReadStmt(TreeNode\* node) {

string id\_name = node->N2->N1->val.second;

string id\_type;

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

if (symboltable[i].first.first == id\_name) {

id\_type = symboltable[i].first.second;

}

}

// 初始化缓冲区，假设 buffer 是一个已经分配了空间的字符数组

emit("# 读入");

emit("la $t0, input\_stack");

emit("addiu $t0, $t0, 1024");

emit("sw $t0, input\_buffer");

if (id\_type == "int") {

// 系统调用5用于读取一个整数

emit("li $v0, 5");

emit("syscall");

emit("sw $v0, " + id\_name); // 将读取的整数保存到变量中

}

else {

// 系统调用8用于读取一个字符串

emit("li $v0, 8");

emit("syscall");

string set\_true = "set\_true" + std::to\_string(label\_id);

string end\_set = "end\_set" + std::to\_string(label\_id);

string set\_false = "set\_false" + std::to\_string(label\_id);

label\_id++;

// 检查字符串的首字符是否为't'或'T'

emit("lb $t0, 0($a0)"); // 读取第一个字符

emit("li $t1, 't'"); // 将 't' 载入到 $t1 寄存器

emit("li $t2, 'T'"); // 将 'T' 载入到 $t2 寄存器

emit("beq $t0, $t1, "+set\_true); // 如果字符等于 't'，跳转到 set\_true 标签

emit("beq $t0, $t2, "+set\_true); // 如果字符等于 'T'，跳转到 set\_true 标签

emit("j "+set\_false); // 如果字符既不等于 't' 也不等于 'T'，跳转到 set\_false 标签

emit(set\_true+":"); // set\_true 标签

emit("li $t0, 1"); // 如果字符串是"true"，设置变量为1

emit("j "+end\_set); // 跳转到 end\_set 标签

emit(set\_false+":"); // set\_false 标签

emit("li $t0, 0"); // 如果字符串不是"true"，设置变量为0

emit(end\_set+":"); // end\_set 标签

emit("sw $t0, " + id\_name); // 将结果保存到变量中

}

}

void processWriteStmt(TreeNode\* node) {

string id\_type;

string id\_name = node->N2->N1->val.second;

//查找符号表以确定类型

emit("# 打印");

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

if (symboltable[i].first.first == id\_name) {

id\_type = symboltable[i].first.second;

break;

}

}

// 负责加载变量到寄存器 $t0 中

if(id\_type=="int")

emit("lw $t0, " + id\_name);

else

emit("lb $t0, " + id\_name);

// 输出的整数和布尔值的处理方式可能有所不同，因此我们在这里进行区分。

if (id\_type == "int") {

emit("li $v0, 1"); // 系统调用代码1表示打印整数

emit("move $a0, $t0"); // 移动我们的整数到 $a0

emit("syscall"); // 进行系统调用

}

else if (id\_type == "bool") {

// 布尔值可以表示为整数0和1，也可以使用某种打印布尔值的方法处理

emit("li $v0, 1"); // 系统调用代码1表示打印整数

emit("move $a0, $t0"); // 移动我们的布尔值到 $a0

emit("syscall"); // 进行系统调用

}

}

// 处理表达式

void processEXPR(TreeNode\* node) {

processTERM(node->N1); // 处理TERM

processADD\_(node->N2); // 处理ADD'

}

// 处理ADD'表达式，对应于非终结符 ADD' → ADD TERM ADD'

// ADD' → empty

void processADD\_(TreeNode\* node) {

if (node== nullptr||node->N1->val.first=="empty") {

// 如果 ADD' → empty, 则不进行任何操作

return;

}

// 处理 TERM

processTERM(node->N2);

// 将栈顶的两个元素出栈进行运算，并将结果压入栈中

emit("# 运算取两个栈顶");

int\_stack\_pop\_1\_2();

if (node->N1->N1->val.second == "+") {

emit("# 加法");

emit("addu $t1, $t1, $t2");

}

else {

emit("# 减法");

emit("subu $t1, $t2, $t1");

}

int\_stack\_push();

// 继续处理 ADD'

processADD\_(node->N3);

}

// 处理TERM，对应于非终结符 TERM → NEGA MUL'

void processTERM(TreeNode\* node) {

processNEGA(node->N1); // 处理NEGA

processMUL\_(node->N2); // 处理MUL'

}

// 处理MUL'表达式，对应于非终结符 MUL' → MUL NEGA MUL'

// MUL' → empty

void processMUL\_(TreeNode\* node) {

if (node== nullptr||node->N1->val.first=="empty") {

// 如果 MUL' → empty, 则不进行任何操作

return;

}

// 处理 NEGA

processNEGA(node->N2);

// 将栈顶的两个元素出栈进行运算，并将结果压入栈中

int\_stack\_pop\_1\_2();

if (node->N1->N1->val.second == "\*") {

emit("# 乘法");

emit("mul $t1, $t1, $t2");

}

else {

emit("# 除法");

emit("div $t1, $t2, $t1");

}

emit("# 将栈顶元素写回");

int\_stack\_push();

// 继续处理 MUL'

processMUL\_(node->N3);

}

// 处理NEGA，对应于非终结符 NEGA → FACTOR

// NEGA → - FACTOR

void processNEGA(TreeNode\* node) {

// Step 1: Process FACTOR

if (node->N1->val.second == "-") {

processFACTOR(node->N2); // 处理 FACTOR

}

else {

processFACTOR(node->N1); // 处理 FACTOR

}

// Step 3: If necessary, negate

if (node->N1->val.second == "-") {

// Step 2: Int\_buffer 出栈到 $t1

int\_stack\_pop\_1();

emit("# 取反");

emit("negu $t1, $t1"); // If it's a negative number, negate the value in $t1

// Step 4: Push the result back to the Int\_num stack

emit("# 写回");

int\_stack\_push();

}

}

// 处理FACTOR，对应于非终结符 FACTOR → ( EXPR )

// FACTOR → id

// FACTOR → number

void processFACTOR(TreeNode\* node) {

if (node->N1->val.second == "(") {

processEXPR(node->N2); // 如果FACTOR是一个括号表达式，处理EXPR

}

else if (node->N1->val.first == "id") {

string id\_name = node->N1->N1->val.second;

string id\_type;

// Search symbol table

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

if (symboltable[i].first.first == id\_name) {

id\_type = symboltable[i].first.second;

break; // Once found, break the loop

}

}

// Load the address of the top of the stack into $t0

string cur\_num;

if (id\_type == "int") {

emit("lw $t1, " + id\_name);

int\_stack\_push();

}

else {

emit("lb $t1, " + id\_name);

bool\_stack\_push\_1();

}

}

else {

string num\_val = node->N1->N1->val.second;

emit("li $t1, " + num\_val);

int\_stack\_push();

}

}

// 处理BOOL，对应于非终结符 BOOL → JOIN BOOL'

void processBOOL(TreeNode\* node) {

processJOIN(node->N1); // 处理JOIN

processBOOL\_(node->N2); // 处理BOOL'

}

// 处理BOOL'表达式，对应于非终结符 BOOL' → || JOIN BOOL'

// BOOL' → empty

void processBOOL\_(TreeNode\* node) {

if (node== nullptr||node->N1->val.first=="empty") {

return; // 如果当前节点为空，则返回

}

processJOIN(node->N2); // 处理JOIN

processBOOL\_(node->N3); // 递归处理BOOL'

bool\_stack\_pop\_1\_2();

emit("andi $t1, $t1, 1");

emit("or $t1, $t1, $t2"); // 执行逻辑或操作

bool\_stack\_push\_1();

}

// 处理JOIN，对应于非终结符 JOIN → NOT JOIN'

void processJOIN(TreeNode\* node) {

if (node == nullptr)return;

processNOT(node->N1); // 处理NOT

processJOIN\_(node->N2); // 处理JOIN'

}

// 处理JOIN'表达式，对应于非终结符 JOIN' → && NOT JOIN'

// JOIN' → empty

void processJOIN\_(TreeNode\* node) {

if (node== nullptr||node->N1->val.first=="empty") {

return; // 如果当前节点为空，则返回

}

processNOT(node->N2); // 处理NOT

processJOIN\_(node->N3); // 递归处理JOIN'

bool\_stack\_pop\_1\_2();

emit("andi $t1, $t1, 1");

emit("and $t1, $t1, $t2"); // 执行逻辑与操作

bool\_stack\_push\_1();

}

// 处理NOT，对应于非终结符 NOT → REL

// NOT → ! REL

void processNOT(TreeNode\* node) {

if (node->N1->val.second == "!") {

processREL(node->N2); // 处理REL

bool\_stack\_pop\_1();

emit("andi $t1, $t1, 1");

emit("xori $t1, $t1, 1"); // 如果是逻辑非操作，对寄存器t0执行取反操作

bool\_stack\_push\_1();

}

else {

processREL(node->N1); // 处理REL

}

}

// 处理REL，对应于非终结符 REL → EXPR ROP EXPR

void processREL(TreeNode\* node) {

processEXPR(node->N1); // 处理EXPR

processEXPR(node->N3); // 处理下一个EXPR

int\_stack\_pop\_1\_2();

string operation = node->N2->N1->val.second;

// 根据运算符生成相应的汇编指令

if (operation == ">") {

emit("sgt $t1, $t2, $t1"); // 对应 > 操作

}

else if (operation == ">=") {

emit("sge $t1, $t2, $t1"); // 对应 >= 操作

}

else if (operation == "<") {

emit("slt $t1, $t2, $t1"); // 对应 < 操作

}

else if (operation == "<=") {

emit("sle $t1, $t2, $t1"); // 对应 <= 操作

}

else if (operation == "==") {

emit("seq $t1, $t2, $t1"); // 对应 == 操作

}

else if (operation == "!=") {

emit("sne $t1, $t2, $t1"); // 对应 != 操作

}

bool\_stack\_push\_1();

}