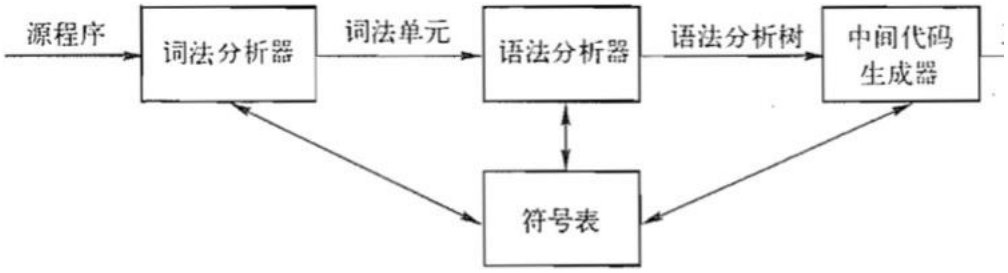
目录

一、概述

1.1 设计要求

一个简单的 C 语言编译器的设计与实现。主要包括词法分析-语法分析-语义分析-中间代码生成-代码优化。具体包含如下内容：
(1)定义一个简单程序设计语言文法(包括变量说明语句、算术运算表
(2)达式、赋值语句;扩展包括逻辑运算表达式、If 语句、While 语句等);
(3)扫描器（词法分析器）设计实现;
(4)语法分析器设计实现;
(5)符号表设计及其生成器的设计与实现;
(6)中间代码设计及其生成器设计实现;
(7)中间代码优化及实现;
(8)目标代码设计及其生成器设计实现;
可以使用扫描器、语法分析器的自动生成器工具。



[图]1.1 一个编译器前端的模型

1.2 符号定义

1.2.1 文法定义

C 语言的 BNF 文法定义：

S -> program
program -> declaration-list
declaration-list -> declaration-list declaration | declaration
declaration -> var-declaration | fun-declaration
var-declaration -> type-specifier ID ; | type-specifier ID [NUM] ;
type-specifier -> int | void
fun-declaration -> type-specifier IDF (params) | compound-stmt
params -> params-list | void
params-list -> params-list , param | param

param -> type-specifier ID | type-specifier ID [ ]
compound-stmt -> { local-declarations statement-list }
local-declarations -> local-declarations var-declaration | empty
statement-list -> statement-list statement | empty
statement -> expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt
expression-stmt -> expression ; | ;
selection-stmt -> if (expression) statement | if (expression) statement else statement
iteration-stmt -> while (expression) statement
return-stmt -> return ; | return expression ;
expression -> var1 = expression | simple-expression
var -> ID | ID [expression]
var1 -> ID1 | ID1 [expression]
simple-expression -> additive-expression relop additive-expression | additive-expression
relop -> <= | < | > | >= | == | !=
additive-expression -> additive-expression addop term | term
addop -> + | -
term -> term mulop factor | factor
mulop -> * | /
factor -> (expression) | var | call | NUM
call -> IDF (args)
args -> arg-list | empty
arg-list -> arg-list , expression | expression

1.2.2 单词种别定义

(1)关键字

else if int return void while

(2)运算符

+ - * / < <= > >= == != = ; , ( ) [ ] { } /* */

(3)标识符

ID=letter letter*
letter= a|..|z|A|..|Z
标识符区别大小写

(4)数字

NUM=digit digit*
digit=0|..|9

(5)空白

具体包含空格、换行符、制表符三种符号

(6)注释

由/*和*/一对组合符号构成。可跨越多行，不能嵌套。

1.2.3 种别码映射表

(1)构成

标记	含义	单词符号
COP	比较操作符	< <= > >= == !=
AOP	赋值操作符	=
OOP	运算操作符	+ - * /
EOP	句末操作符	;
SOP	结构分隔符	( ) , [ ] { }
RESERVED	保留字	int if else return void while

[表]1.2.3(1) 种别码构成

(2)单词-种别码映射表

种别码 syn	单词符号 symbol	类型 type	种别码 syn	单词符号 symbol	类型 type
0	#	end	15	>=	COP
1	else	RESERVED	16	==	COP
2	if	RESERVED	17	!=	COP
3	int	RESERVED	18	(	SOP
4	return	RESERVED	19	)	SOP
5	void	RESERVED	20	[	SOP

6	while	RESERVED	21	]	SOP
7	+	OOP	22	;	EOP
8	-	OOP	23	,	SOP
9	*	OOP	24	{	SOP
10	/	OOP	25	}	SOP
11	<	COP	26	letter|letter*	id
12	>	COP	27	digit|digit*	num
13	=	COP			
14	<=	COP			

[表]1.2.3(2) 单词-种别码映射表

二、系统设计

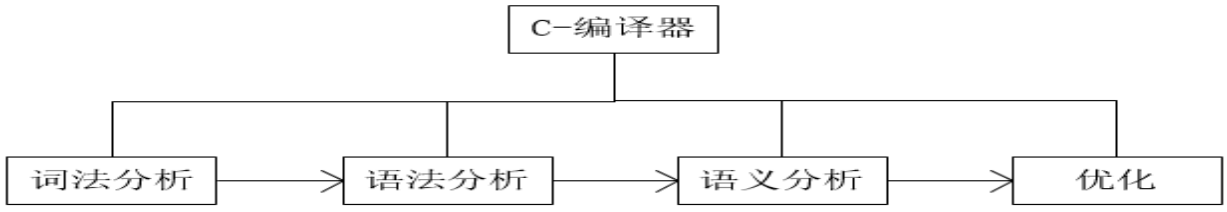
2.1 后端设计

2.1.1 总体处理流程

总体处理程序分为两个主要部分，分别处理源程序和语法规则。

(1)对源程序的处理包含如下主要步骤：

- 预处理：删除空格、注释
- 词法分析
- 语法分析
- 语义分析
- 中间代码生成
- 代码优化



[图]2.1.1 编译器处理流程

(2)对语法规则的处理包含如下主要步骤：

- 消除左递归、回溯
- 消除”或”符(|)号
- 计算终结符、非终结符

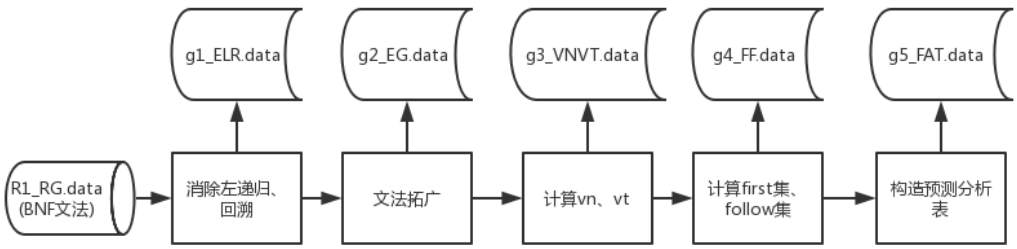
- 求得 first 集，follow 集
- 构造预测分析表

(3)各步骤间的关系

- 由 R0_Project.c 存储源程序，R1_RG.data 存储语法规则
- 程序读入两个文件后，首先进行预处理操作，对源程序去除注释，去除多余空格，对语法规则消除左递归以及公共左因子，并且根据|拆分语法推导式，根据后序语法分析发现的问题后回过来对语法做的一些细微的修改也在这一步完成。
- 随后对处理过后的源程序进行词法分析，得到一个 tokne 流，供语法分析时使用，生成栈以及语法分析树。
- 对处理后的语法规则，首先计算出所有的终结符以及非终结符，并求出非终结符的 first,follow 集，生成预测分析表。
- 根据 token 流计算出符号表
- 根据 token 流以及预测分析表完成整个程序的出入栈以及语法树的建立
- 使用深度遍历语法树遇到相应节点，即进行语义动作，产生三地址代码。
- 最后对产生额三地址代码进行优化。

2.1.2 文法处理

(1)文法处理流程图



[图]2.1.2(1)文法处理流程

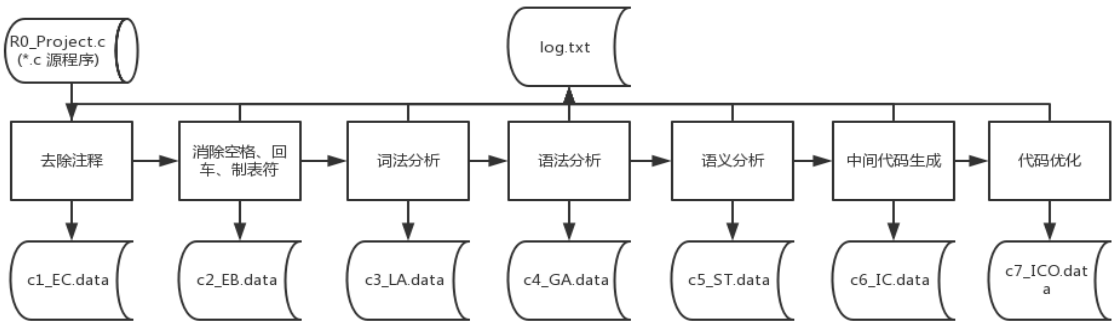
(2)流程说明

- 文法处理程序首先读入 R1_RG.data 文件，R1_RG.data 文件里面存放的是 C 语言的 BNF 文法。
- 然后消除文法中的左递归和回溯，并将处理后的文法输出到 g1_ELR.data 文件；
- 随后对文法以或(|)为分隔符进行分割拓广。并将拓广后的文法输出到 g2_EG.data 文件；

·与此同时，计算出文法中的终结符 vt 和非终结符 vn，目的是为了下一步计算 first 集和 follow 集。并将求得的终结符和非终结符输出到 g3_VNVT.data 文件；
·进而求出文法的 first 集和 follow 集，并将结果输出到 g4_FF.data 文件；
·最后，根据求得的 first 集和 follow 集，构造出预测分析表。并将预测分析表输出到 g5_FAT.data 文件。至此，对文法的处理结束。

2.1.3 编译处理

(1)编译处理流程图



[图]2.1.3(1)编译处理流程

(2)流程说明

·编译处理程序首先读入 C 语言源程序 R0_Project.c 文件，对程序进行简单的预处理。包括去除注释，消除回车、空格、制表符。
·然后进行词法分析-语法分析-语义分析-中间代码生成-代码优化。
·每执行一步骤，都会把每一步的结果保存为*.data 文件输出出来，并将每一步的执行过程写入到日志文件 log.txt。以便前端接口调用。

2.2 前端设计

2.2.1 功能

前端主要能够对 C 语言工程文件进行读写编辑操作、对 C 语言程序进行编译，并将编译的每一步实时的在前端控制台输出出来。出现的错误能够提示出具体的位置。

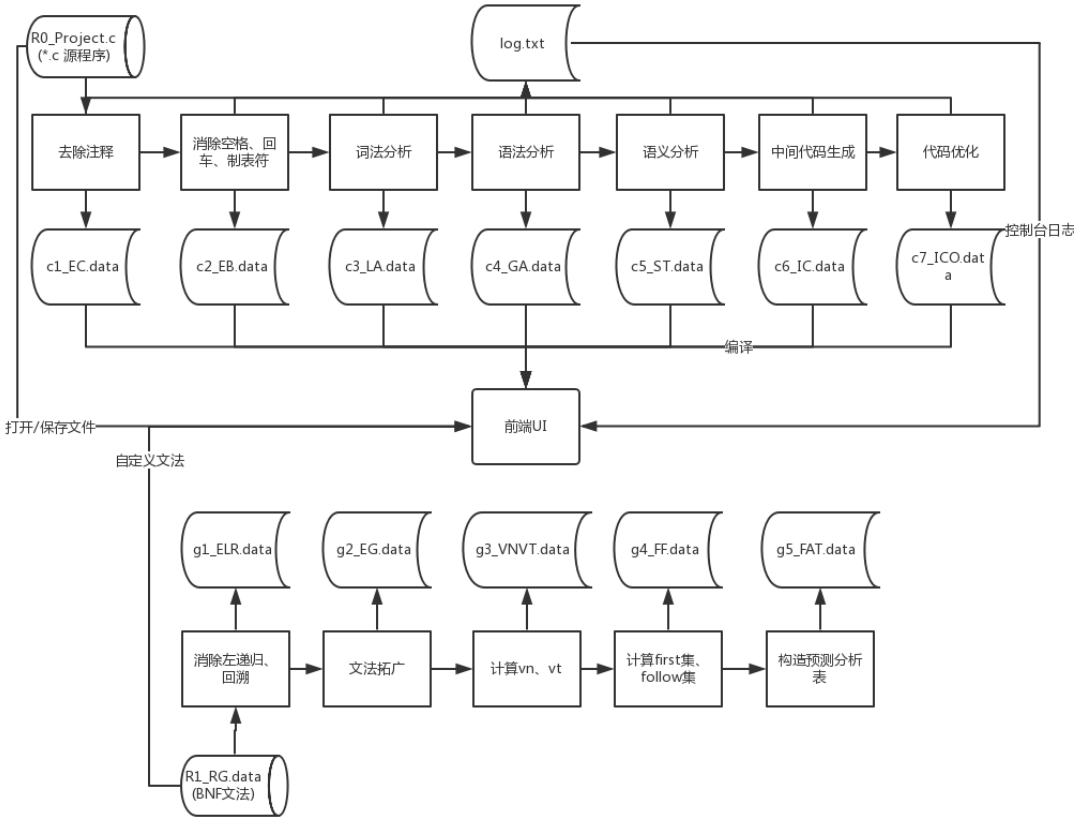
2.2.2 实现方式

前端 UI 使用 Java GUI 方式实现。通过后端预留的文件接口与 Java GUI 进行交互。具体的每一个前端按钮对应一个后端输出的*.data 或*.c 或*.log 类型的文件。通过点击前端按钮，来对这些后端输出的文件进行读写操作，达到前后端交互的目的。后端输出的*.data、*.c
*.log 文件是前后端功能交互的主要接口。

具体接口名称及对应的前端功能，见章节【2.3 接口】部分。

2.3 接口

前后端交互接口图：



[图]2.3 交互接口

三、系统实现

3.1 定义

3.1.1 结构体

(1)变量元素结构体

```
struct symbol
{
    string name;      //变量名
    string type;      //变量类型
    string property; //变量属性
    struct symbol *st;   //函数型成员地址
};
```

定义于 symbol.cpp，在计算符号表时存储函数获取的形参、全局变量以及其中定义的局部变量。

(2)函数结构体

```
struct func
{
    string name;      //函数名
    string ret;          //函数返回值
    int num;          //函数中参数个数
    struct symbol st[255];  //函数符号表
};
```

定义于 symbol.cpp，在计算符号表时存储各函数符号表以及各类信息。

(3)token 结构体

```
typedef struct Token
{
    string re;        //token 原始字符串
    string type;      //token 类型
    int line;         //token 所在行
    int colume;          //token 所在列
}Token;
```

定义于 global.cpp，适用整个工程，源程序由词法分析后产生的 token 流中所使用的结构体。

(4)节点语义信息结构体

```
typedef struct info
{
    int arrayflag;        //记录变量是否为数组
    int emptyflag;        //记录该非终结符是否推出空
    int whilenum1;        //记录 while 入口的三地址代码值
    int whilenum2;        //记录 while 出口的三地址代码值
    int ifnum1;           //记录 if 语句真出口三地址代码值
    int ifnum2;           //记录 if 语句假出口三地址代码值
    int elsenum1;     //记录 else 入口三地址代码值
    int elsenum2;     //记录 else 出口三地址代码值
    int returnflag;
}info;
```

定义于 global.cpp，适用整个工程，用于存储由源程序生成的语法分析树中各节点的语义信息。

(5)语法分析树节点结构体

```
typedef struct Gtree
{
    char re[255];     //结点元素原字符串
    char type[255];       //结点元素类型    非终结符为 derivation  终结符为自身类型 根节点为 head
    int bitnum;           //结点孩子节点个数
    int mtnum;        //如为非终结符 记录推导式编号
    int nodenum;
    struct Gtree * sons[255];       //孩子节点地址数组
    struct Gtree *brother;          //兄弟节点地址
    struct Gtree *father;           //父亲节点地址
    struct info *info;              //语义信息
}treebit, *bit;
```

定义于 global.cpp，适用整个工程，用于构建源程序生成的语法分析树。

(6)栈结构体

```
 typedef struct Stack
{
    vector<string> st;       //栈
    int current;          //当前栈中元素个数
}st;
```

定义于 global.cpp，适用整个工程，用于 lll 语法分析时的出入栈操作。

(7)预测分析表结构体

```
typedef struct Mtable
{
    map<string, int>vnname;      //非终结符表
    map<string, int>vtname;      //终结符表
    int mtable[maxsize][maxsize];       //预测分析表
    int vnnum;     //记录非终结符个数
    int vtnum;     //记录终结符个数
}mt;
```

定义于 global.cpp，适用整个工程，用于存储 111 语法的预测分析表。

3.1.2 全局变量

(1)Global.h 中全局变量

```
static string resource = "R0_Project.c";//待编译源程序
static string file_grammar = "R1_RG.data";    // 源 文 法 (Resource Grammmer)文件,BNF 文法
static string file_grammar = "R2_WT.data";    //单词种别映射表(Word type)
static string file_lllgrammar = "g1_ELR.data";   // 消 除 左 递 归 (Eliminate left recursion)，回溯
static string file_ad_grammar = "g2_EG.data";// 拓 广 文 法 (Extension grammar)
static string file_vn_vt = "g3_VNVT.data";    //终结符,非终结符存储文件
static string file_ff = "g4_FF.data"; //first,follow 集存储文件
static string file_mtable = "g5_FAT.data";       // 预 测 分 析 表 (Forecast analysis table)存储文件
static string file_nozs = "c1_EC.data";   // 去 注 释 后 (Eliminate comments)的文件
static string file_clear = "c2_EB.data";  //消除空格及其空白字符(Eliminate blanks)
static string file_lex = "c3_LA.data";    //词法分析(lexical analysis)结果存储文件
static string file_showstack = "c4_GA.data"; // 语 法 分 析 (Gramma analysis)，语法分析栈内信息输出
static string file_symtable = "c5_ST.data";  //符号表(Symbol table)存储文件
static string file_midcode = "c6_IC.data";       //  中  间  代  码 (Intermediate code)输出
```

```
static string file_optimize = "c7_ICO.data"; // 中 间 代 码 优 化 (Intermediate code optimization)
static string file_log = "log.txt";          //进程日志
static string period[7] = { "",//执行阶段名称（总步骤名称列表）
                          "1:预处理/初始化",
                          "2:词法分析      ",
                          "3:语法/语义分析",
                          "4:中间代码生成 "
};
static string content1[10] = { "",//各阶段各步骤的目标输出名称（分步名称列表）
                             "去除注释",
                             "去除空白字符",
                             "消除左递归以及公共左因子",
                             "将语法根据|拆分",
                             "求 vn,vt 集合",
                             "求 fisrt,follow 集",
                             "计算预测分析表 mtable"
};
static string content2 = "词法分析";
static string content3[10] = { "",
                             "生成语法分析树",
                             "生成符号表",
};
static string content4[3] = { "",
                            "中间代码生成",
                            "中间代码优化"
};
static string call = "call ";
static string param = "param ";
static string proc = "Prog ";
static int maxline;
static const int maxsize = 255;
static const string firststr = "S";
static fstream in, out;
static map<string, string> vttype; //终结符类型表
static vector<string> vn;          //非终结符集合
static vector<string> vt;          //终结符集合
//各类要使用到的存储器指针
static vector<string>::iterator iter;
static map<string, string>::iterator iter_map;
static map<string, vector<string>>::iterator maper;
static map<string, int>::iterator map_si;
static map<int, vector<string>> grammarmap;      //存储语法的推导
```

```
static map<int, vector<string>>::iterator gmaper;
static map<string, vector<string>> firstlist;    //first 集
static map<string, vector<string>> followlist;      //follow 集
static map<string, int> emptymake;        //存储所有非终结符能推导出空字的推导式编号
```

(2)Grammar_Analysis.cpp 中全局变量

```
Token token;
struct Mtable mtable;
struct Stack stack;
treebit *treenode;   //语法分析树根节点
int nodenum = 0;     //节点编号
map<int, vector<string>> placelistmap;    //各节点存储字符串信息表
map<string, bool(*)(treebit *cur)> funcmap;      //推导式编号与其对应的语义动作函数
map<int, string> midcodemap;
map<string, int> funcnum;
int curnum = 99;     //三地址代码编号
int curtemp = 0;     //临时变量个数
```

(3) Symbol_Table.cpp 中全局变量

```
struct func functable[255];     //符号表
int funcnum = 0;            //符号表中元素个数
```

(4) Intermediate_Code_Optimization.cpp 中全局变量

```
map<int, vector<string>> gb;       //中间代码表
map<string, double>::iterator con;
vector<int> entry;          //基本块入口地址表
vector<int>::iterator it,itertemp;
int endline;  //中间代码的最后一条
map<string, int> op;
```

3.1.3 函数

(1)Global 全局函数

```
//打开指定文件输入/输出 返回 1 表示打开成功，返回 0 表示打开失败
int infile(fstream &in, string filenmae);
int outfile(fstream &out, string filenmae);
//关闭 in,out 两个用来输入输出的文件
void closefile(fstream &file1,fstream &file2);
//输出文件内容
void readfile(string filename);
//分割线
void divideline(void);
//各过程处理以及界面输出
void precedure(string period, int num, string filein, string fileout,
void(*func)(fstream &, fstream &),string content);
//去除字符串前后多余的空格
void fix(string &str);
```

(2)Lexical_Analysis

```
int isBC(char ch);       //判断是否为空白字符
int Reserve(string strToken);
void Retract(fstream &source);
void lexicalanalysis(fstream &in, fstream &out); //词法分析
void fixlex(void);       //修改文法
```

(3)Intermediate_Code_Optimization

```
//读入所有三地址代码
void getbetter1(fstream &in);
//常量优化
void getbetter2(int start, int end);
//优化中间代码
void optimize(fstream &in, fstream &out);
```

(4)Pretreatment

```
//删除程序中所有注释
void delzs(fstream &in, fstream &out);
//删除程序中所有多余的空格,tab 以及回车
void delblank(fstream &in, fstream &out);
//消除左递归以及公共左因子
void delleftrecursion(fstream &in, fstream &out);
//将语法根据|拆分
void splitgrammar(fstream &in, fstream &out);
```

(5)Symbol_Table

```
//输出符号表
void output(fstream &out);
//生成符号表
void calsymboltable(fstream &in, fstream &out);
//判断函数是否已经存在 并返回在该函数在函数名表中的位置 不存在则返回-
1
int func_exist(struct func functable[255], int funcnum, string str);
//判断变量是否出现在当前函数的符号表中 存在则返回表中位置 否则返回-1
int sym_exist(struct func functable[255], struct symbol *st, int num,
string str);
//将当前函数名加入到当前函数表中
void pushsym(struct func functable[255], struct symbol *st, int curnum,
int num);
//将当前函数加入到函数表中 并返回该函数符号表的地址
struct symbol * pushfunc(struct func functable[255], int &funcnum,
string str, string ret);
//初始化符号表
void init_symtable(void);
//将当前变量加入到当前函数表中
void pushsym2(struct func functable[255], struct symbol *st, int curnum,
int num, string name, string property, string type);
```

(6)Grammar_Analysis

```
//初始化终结符类型表
void init_vttype(map<string, string> &vttype);
//初始化预测分析表 mtable
```

```
void init_mtable(struct Mtable &mtable);
//判断集合中是否存在 str 元素
int check_exist(vector<string> vec, string str);
//判断 vtmap 中是否有该 vt
int checkmap(map<string, string> mp, string str);
//删除 vt 集中应该为 vn 的元素
void del(vector<string> &vec, string str);
//计算终结符集合以及非终结符集合
void cal_vn_vt(fstream &in, fstream &out);
//输出 vector 中所有信息 输出 vn,vt 信息至文件
void output_vec(fstream &out);
//求 first 集
void cal_first();
//求 follow 集
void cal_follow();
//计算 first follow 集
void cal_first_follow(fstream &in, fstream &out);
//输出 first,follow 集至文件
void output_ff(fstream &out);
//计算预测分析表
void cal_mtable(fstream &in, fstream &out);
//初始化栈
void stack_init(struct Stack &stack);
//元素入栈
void stack_pushin(struct Stack &stack, string str);
//获取栈顶元素
string stack_gettop(struct Stack stack);
//栈顶元素出栈
void stack_popout(struct Stack &stack);
//输出栈中信息
void stack_show(struct Stack stack, fstream &out);
//初始化语法树
void tree_init();
//生成语法分析树
void make_tree(fstream &in, fstream &out);
//输出中间代码
void printmidcode(fstream &out);
//初始化语义动作表
void init_funcmap();
//中间代码生成
void midcode(fstream &in, fstream &out);
//语义动作函数
bool fomula_1(treebit *temp);
…
```

bool fomula_82(treebit *temp);

3.2 后端主要函数功能

3.2.1 main

(1)void precedure(string period, int num, string filein, string fileout, void(*func)(fstream &, fstream &),string content);

获取第一、二个参数为当前过程名称及过程中的第几步，第三、四个参数获取获取的文件名以及输出的目标文件名，第五个参数为进行该步操作所需要调用的函数名称，分别定义在了各分块的文件中，最后一个参数为该步骤所进行为何操作。

函数中首先打开了输入输出文件并判断是否打开成功，随后调用获取的目标函数，将两个文件流操作变量传递给它。输出当前步骤的各种信息以及各文件名作为提示。函数最后将两个文件都关闭。

函数中调用了打开输入文件函数 infile,打开输出文件函数 outfile,关闭文件函数 closefile 以及输出分割线函数 divideline。

3.2.2 Pretreatment

(1)void delzs(fstream &in, fstream &out);void delblank(fstream &in, fstream &out);

函数主体均由 switch-case 语句构成，来实现有限状态自动机，详细的状态转换见 pretreat.h，消除多余空白符时，会记录前一个出现的空白字符，如果为回车则会永久记录下来，保证尽量不改变代码换行的位置。去除注释时，在判断一段文字是否为注释时，使用一个字符串记录还未被确定不是注释的内容，以防在文件末尾注释右侧未封闭导致出现代码的缺失(虽然这种情况下是显然不符合语法规则的代码)。

(2)void delleftrecursion(fstream &in, fstream &out);

函数用于消除语法的左递归以及公共左因子，其中首先将一个推导式以字符串读入，根据->的位置将字符串进行拆分，记录左侧字符串为 part1，右侧为 part2。下一步将 part2 根据|拆分为多个部分，该操作调用函数 splitblank，返回一个数组其中存储了各部分的字符串值和一个 int 值表示有几个部分。下一步由于观察语法推导式可以发现所有推导式出现左递归的情况只会出现在右侧第一个推导，所以只判断第一部分是否出现了左递归，如果出现了则按照 A->Ab|c A->cA’和 A’->bA’|empty。随后判断公共左因子，也很容易发现存在公共左因子的情况也只会出现在右侧只有两个部分时，那么也可以简化算法，判断前两个是否出现了公共的左因子，也按照类似左递归进行消除。

(3)void splitgrammar(fstream &in, fstream &out)

函数中将已经适应 111 规则的语法规则每一条根据|进行拆分，并在每一条推导式前添加编号进行计数。

3.2.3 Grammar_Analysis

(1)void init_vttype(map<string, string> &vttype)

初始化终结符类型表使用 string-string 类型的 map 表，键为原始终结符符号，值为其类型，具体分类见 grammar-analysis.h 中定义。

(2)void init_mtable(struct Mtable &mtable)

初始化预测分析表使用专用定义的结构体，初始化时将预测分析表的第一列填充为所有非终结符，第一行填充为所有终结符，并且所有位置预设初始值 0 即不能推导。

(3)void stack_init(struct Stack &stack)

初始化栈时首先压入最终的#以及所有推导式的第一个产生的非终结符，即默认程序一开始已经读入一个#并且出栈(默认起始入栈#[firststr]#)

(4)void tree_init()

初始化语法分析树，首先构建根节点 treenode，给它三个分支# [firststr] #，将所有节点信息预设赋值。

(5)void init_funcmap()

初始化语义动作表，每一条语法产生式都对应着一个语义动作函数，以键值对$[num]-fomula_[num]定义，有特殊的推导式例如 if 以及 while 产生式需要多个不同位置的语义动作，则添加多个定义。

(6)void cal_vn_vt(fstream &in,fstream &out)

该函数计算出所有的非终结符以及终结符，函数中逐条读入推导式，根绝->将推导式分为左右两个部分，对于左边部分，判断是否已经出现在非终结符表中，如果没有出现则加入，并判断是否出现在终结符表中，如果出现则从终结符表中删除，对于右边部分，判断是否出现在终结符表中，如果未出现则加入。使用 check_exsist 传入字符串以及表明，返回 1 表示出现，0 表示未出现，使用 del 传入表明以及字符串删除表中对应的字符串。

(7)void cal_first()

该函数计算所有非终结符的 first 集，整体使用 while(1)循环，使用一个 flag，如果整个循环中对于所有的 first 集合没有进行改变则跳出循环。

First、follow 集均采用 string-vector<string>存储，键为非终结符名称，值为存储 first 或 follow 字符串的 vector 容器

- 循环初始 flag=1 逐条分析推导式 推导式使用 int-vector<string>的 map 类型存储，int 表示推导式的编号，vector<string>存储根据空格拆分后推导式。
- 判断推导式第一个元素是否为终结符没如果为终结符则判断该终结符是否在当前非终结符的 first 集中，如果不在则加入并且 flag=0
- 如果第一个元素为非终结符，则将该非终结符的 first 集中的所有元素加入推导式首部的非终结符的 first 集中，且 flag=0
- 如果推导式前部分的非终结符的 first 集中有 empty，则将他们所有的 first 集中的所有元素加入至首部的非终结符的 first 集中，且 flag=0，直到遇到一个不能推导出 empty 的非终结符或是终结符。

(8)void cal_follow()

该函数计算所有非终结符的 follow 集，整体使用 while(1)循环，使用一个

flag，如果整个循环中对于所有的 follow 集合没有进行改变则跳出循环。
·循环初始 flag=1 逐条分析推导式 最开始将#置于[firststr]的 follow 集中
分析推导式时逐个遍历式中元素
·当前元素为终结符，无需操作，进入下一个元素
·当前元素为非终结符，向后遍历直到出现终结符或非终结符不能推导出 empty 时停止，将所有能推出 empty 的非终结符的 first 集中所有元素或终结符加入该非终结符的 follow 集中
·由产生式的最后向前遍历，直到遇到终结符或不能推导 empty 的非终结符停止将推导式首部的非终结符的 follow 集添加至尾部所有能推导出 empty 的非终结符的 follow 集中
(9)void cal_mtable(fstream &in, fstream &out)

该函数计算出预测分析表，遍历推导式表，对于每一条推导式，先取出箭头左侧的非终结符作为预测分析表的行，再取出箭头右侧第一个符号，对于它的所有的 first 集，将该推导式编号填入 first 集内容所对应的列中，如果遇到 empty 将 emptyflag 赋值为 1，循环后，判断推导式左侧的非终结符是否能退出空，如果有，则将那条推导式编号填入该非终结符 follow 集元素所对应的列中。
(10)void make_tree(fstream &in, fstream &out)

该函数在对 token 流读取并根据预测分析表进行出入栈的同时建立语法分析树。
·出入栈操作

当栈顶元素为终结符时，将栈顶元素出栈并判断当前读进的符号是否与出栈元素相同，如果不相同则报错。

当栈顶元素为非终结符时，将此非终结符作为行，读入的符号作为列，查看预测分析表中该行该列的值，如果为 0 则报错，如果不为零，则先将栈顶元素出栈，然后将其值所对应的推导式由后往前一次入栈
·建树操作

首先设置当前节点为 treenode 根节点的 son[2]位置。

每次使用推导式推导时，将推导式所有元素按顺序新建节点插入当前节点的 sons 中，并在推导结束时添加$[num]的语义操作节点。

每次推导结束时进入当前节点的 sons[1]节点，每次出栈时进入当前节点的 brother 节点，如果当前节点无 brother 节点，则进入其 father 节点后再次寻找 brother 节点，知道 brother 节点存在并且不为语义动作节点。
(11)bool fomula_[num] (treebit *temp)

各语义动作节点
(12)void midcode(fstream &in, fstream &out)

该函数负责中间代码生成，使用深度优先遍历遍历整个语法分析树，调用 deepsearch 函数，碰到语义动作节点则调用相对应的语义动作函数。

3.2.4 Symbol_Table

(1)int func_exist(struct func functable[255], int funcnum, string str)

判断当前函数是否存在符号表中，若存在返回该函数在符号表中的位置否则返回-1
(2)int sym_exist(struct func functable[255], struct symbol *st, int

num, string str)

判断当前变量时候存在符号表中,若存在返回该变量在符号表中的位置否则返回-1

(3)void pushsym(struct func functable[255], struct symbol *st, int curnum, int num)

将当前函数名加入到当前函数表中

(4)void pushsym2(struct func functable[255], struct symbol *st, int curnum, int num, string name, string property, string type)

将当前变量名加入到当前函数表中

(5)struct symbol * pushfunc(struct func functable[255], int &funcnum, string str, string ret)

将当前函数加入到函数表中，并返回该函数符号表的地址

(6)void init_symtable(void)

初始化符号表

(7)void output(fstream &out)

输出符号表

(8)void calsymboltable(fstream &in, fstream &out)

生成符号表

- 读入词法分析中的每一个 token
- 遍历所有的 token，在遍历的过程中检测当前 token 属性并作出相应操作。

在检测过程中设置了两个 flag 用于标记大括号和小括号的计数，从而判断当前 token 所处的位置，再结合对应的属性对 token 进行分类处理。

- 如果当前 token 为函数名。函数分为定义和调用的两种情况。

如果该函数名 token 之前不存在左大括号，说明是函数的定义，检查表中是否有该函数，若无该函数则将该函数写入表中。

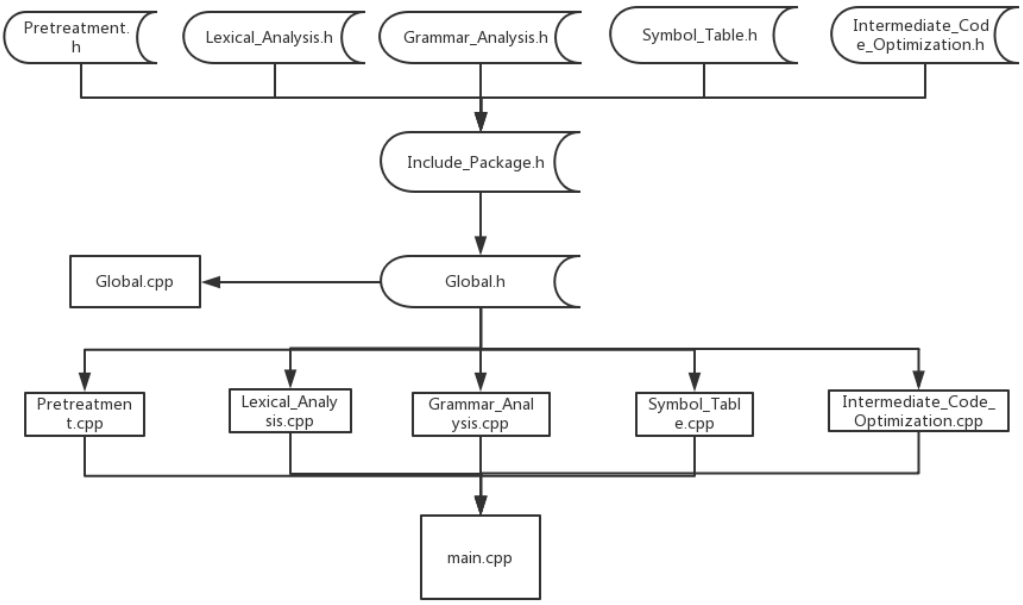
如果该函数名之前存在一个或多个左大括号，则说明这是函数的调用。检查符号表，若表中不存在该函数说明未进行函数声明就调用，进入报错处理；若存在该函数再判断该函数名中的变量是否存在表中，如果是首次调用则将变量加入到函数表中。

- 如果当前 token 为变量名，同样分为变量定义和调用的处理。

如果该变量名前有‘int’或‘void’，则表示是变量的定义，而变量的定义只能存在函数参数的小括号内或者不能存在小括号内部，根据这一条件判断是否合法。若是合法的变量定义再检查表中，若不存在则入表。

如果变量是调用，检查表中是否存在，若不存在，说明变量未经定义就调用，进入报错处理。

3.2.5 工程结构依赖关系图



[图]3.2.5 工程结构依赖

3.3 前端实现

3.3.1UI 构建

```
private void initialize() {
        frame = new JFrame("编译原理课程设计_Compiler");
        frame.setResizable(false);
    frame.setIconImage(Toolkit.getDefaultToolkit().getImage("window.i
co"));
        frame.setBounds(100, 100, 1040, 712);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(null);
        JTextArea textArea_1 = new JTextArea();
        JTextArea textArea_2 = new JTextArea();


        //以下实现 UI 上部编码区
        JPanel panel_3 = new JPanel();
        panel_3.setBounds(10, 37, 1040, 369);
        frame.getContentPane().add(panel_3);
        panel_3.setLayout(null);
        textArea_2.setFont(new Font("仿宋", Font.PLAIN, 24));
```

```
        textArea_2.setBounds(1, 1, 1010, 352);
        panel_3.add(textArea_2);
        JScrollPane scrollPane = new JScrollPane(textArea_2);//实现滚
动条
        scrollPane.setBounds(0, 0, 1015, 352);
        panel_3.add(scrollPane);
        //以下实现顶部操作按钮
        JPanel panel = new JPanel();
        panel.setBounds(0, 0, 1040, 42);
        frame.getContentPane().add(panel);
        panel.setLayout(null);

        JButton btnNewButton = new JButton("打开文件");
        btnNewButton.setBounds(15, 5, 105, 29);
        panel.add(btnNewButton);
        JButton btnNewButton_4 = new JButton("帮助");
        btnNewButton_4.setBounds(762, 5, 69, 29);
        panel.add(btnNewButton_4);      JButton btnNewButton_5 = new
JButton("清除退出");
        btnNewButton_5.setBounds(907, 6, 116, 29);
        panel.add(btnNewButton_5);
         //以下实现 UI 下部编译区
        JPanel panel_1 = new JPanel();
        panel_1.setBounds(10, 460, 1040, 199);
        frame.getContentPane().add(panel_1);
        panel_1.setLayout(null);
        textArea_1.setFont(new Font("等线", Font.PLAIN, 14));
        textArea_1.setBounds(0, 0, 1010, 200);
        panel_1.add(textArea_1);

        JScrollPane scrollPane_1 = new JScrollPane(textArea_1);//实现
滚动条
        scrollPane_1.setBounds(0, 0, 1015, 200);
        panel_1.add(scrollPane_1);
        //以下实现下部 UI 单选按钮
        JPanel panel_2 = new JPanel();
        panel_2.setBounds(0, 405, 1040, 50);
        frame.getContentPane().add(panel_2);
        panel_2.setLayout(null);
        ButtonGroup buttonGroup = new ButtonGroup();//实现单选，构造
ButtonGroup
        JRadioButton rdbtnNewRadioButton = new JRadioButton("实时");
        rdbtnNewRadioButton.setBounds(15, 21, 69, 29);
        panel_2.add(rdbtnNewRadioButton);
```

```
        buttonGroup.add(rdbtnNewRadioButton);
        JRadioButton rdbtnNewRadioButton_3 = new JRadioButton("代码预处理");
        rdbtnNewRadioButton_3.setBounds(195, 21, 123, 29);
        panel_2.add(rdbtnNewRadioButton_3);
        buttonGroup.add(rdbtnNewRadioButton_3);       JRadioButton rdbtnNewRadioButton_1 = new JRadioButton("词法分析");
        rdbtnNewRadioButton_1.setBounds(375, 21, 105, 29);
        panel_2.add(rdbtnNewRadioButton_1);
        buttonGroup.add(rdbtnNewRadioButton_1);
        JRadioButton rdbtnNewRadioButton_2 = new JRadioButton("语法分析");
        rdbtnNewRadioButton_2.setBounds(555, 21, 105, 29);
        panel_2.add(rdbtnNewRadioButton_2);
        buttonGroup.add(rdbtnNewRadioButton_2);
        JRadioButton rdbtnNewRadioButton_4 = new JRadioButton("中间代码生成");
        rdbtnNewRadioButton_4.setBounds(735, 21, 141, 29);
        panel_2.add(rdbtnNewRadioButton_4);
        buttonGroup.add(rdbtnNewRadioButton_4);
        JRadioButton rdbtnNewRadioButton_5 = new JRadioButton("代码优化");
        rdbtnNewRadioButton_5.setBounds(915, 21, 105, 29);
        panel_2.add(rdbtnNewRadioButton_5);
        buttonGroup.add(rdbtnNewRadioButton_5);
    }
```

3.3.2 接口通信映射表

序号	前端按钮	后端接口	功能
1	打开文件	*.c	打开工程文件(*.c 文件)
2	保存文件	*.c	保存工程为*.c 文件
3	编译	Compiler_.c.exe	执行编译
4	自定义文法	R1_RG.data	自定义编译器文法
5	帮助	R2_WT.data	显示帮助信息
6	清除退出	Clean_Cache.bat	清除缓存并退出程序
7	实时	log.txt	显示编译过程的实时状态
8	代码预处理	c2_EB.data	显示预处理后的代码
9	词法分析	c3_LA.data	显示词法分析结果
10	语法分析	c4_GA.data	显示语法分析结果
11	中间代码生成	c6_IC.data	显示中间代码

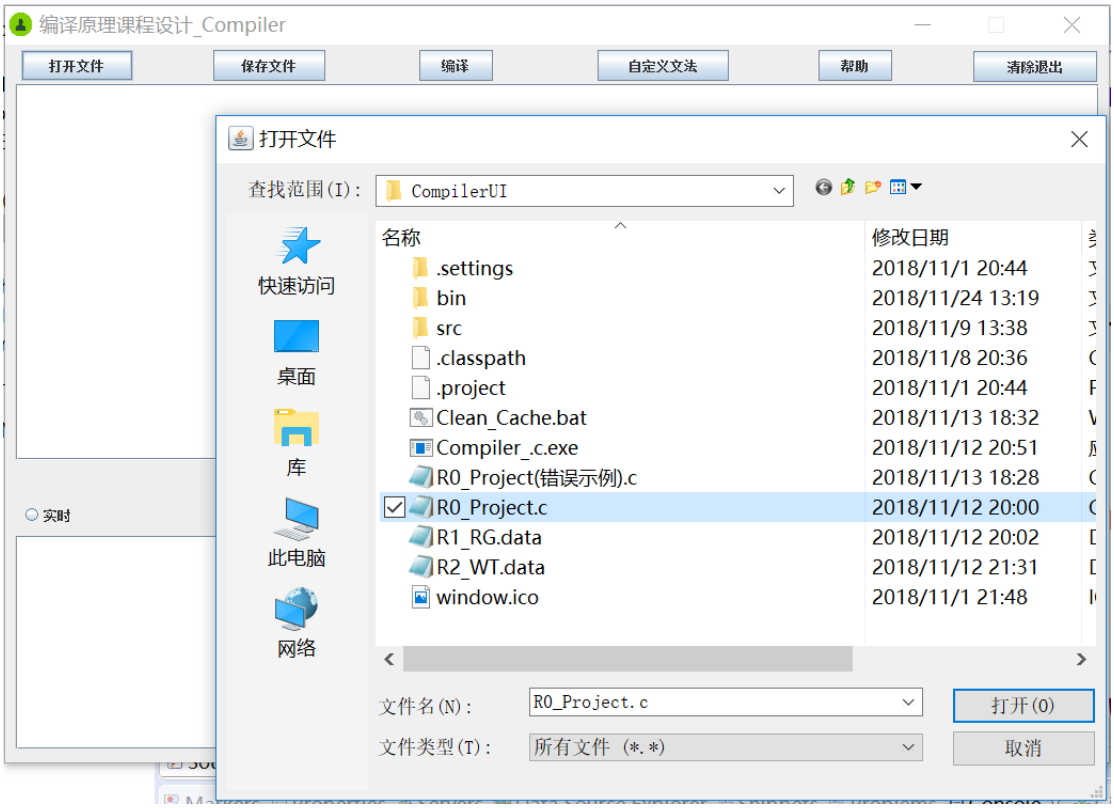
12	代码优化	c7_ICO.data	显示优化后的中间代码

[表]3.3.2 接口通信映射表

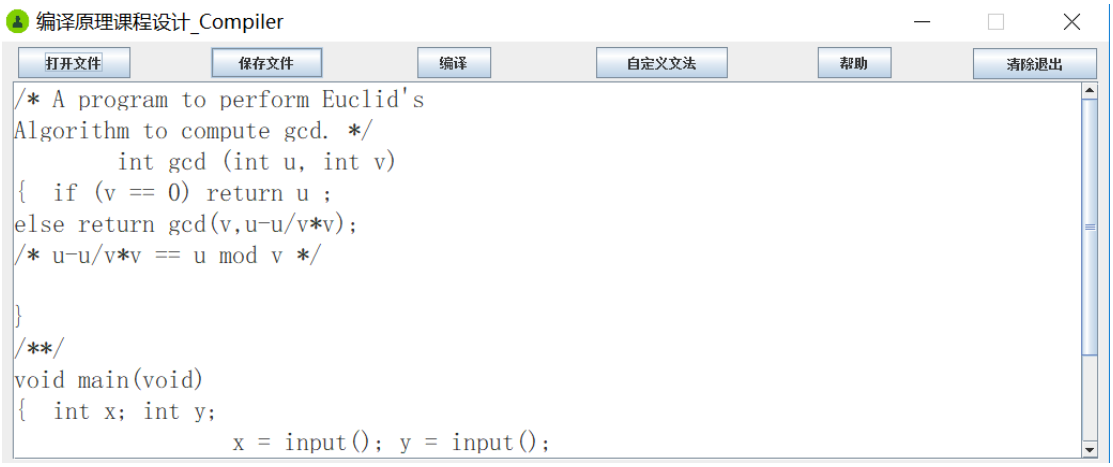
四、集成测试

4.1 文件读写

(1)打开*.c 文件

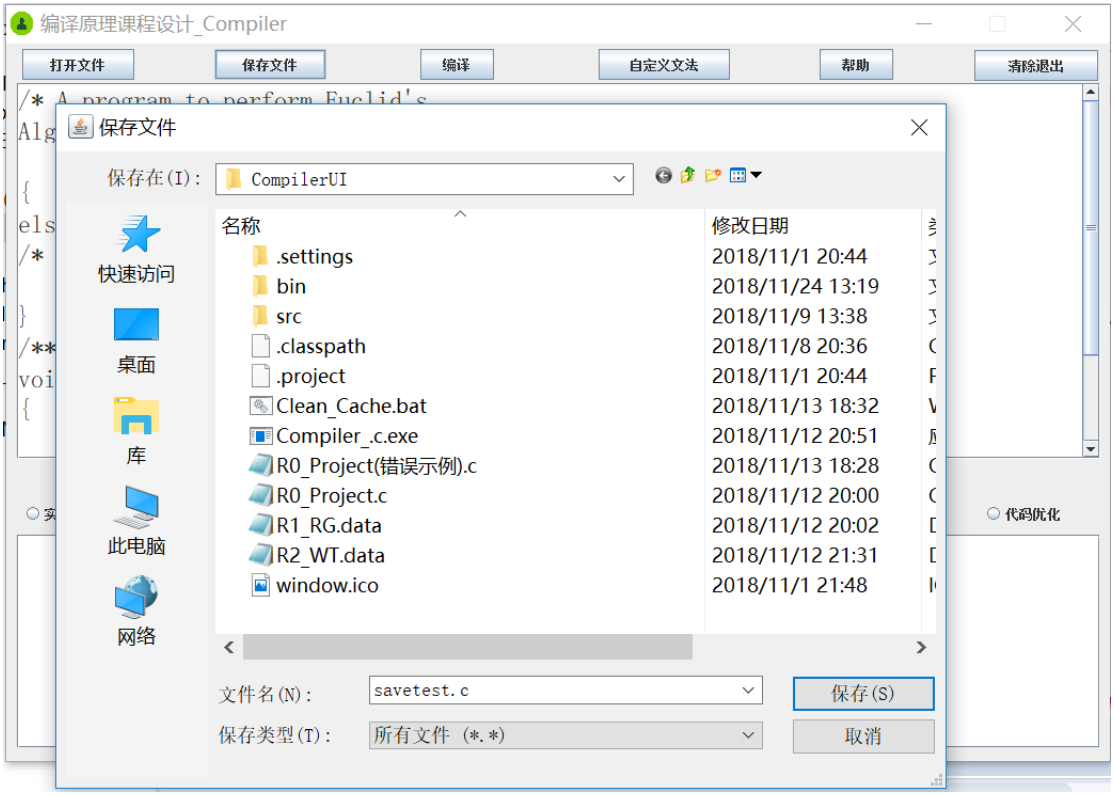


[图]4.1(1)打开文件



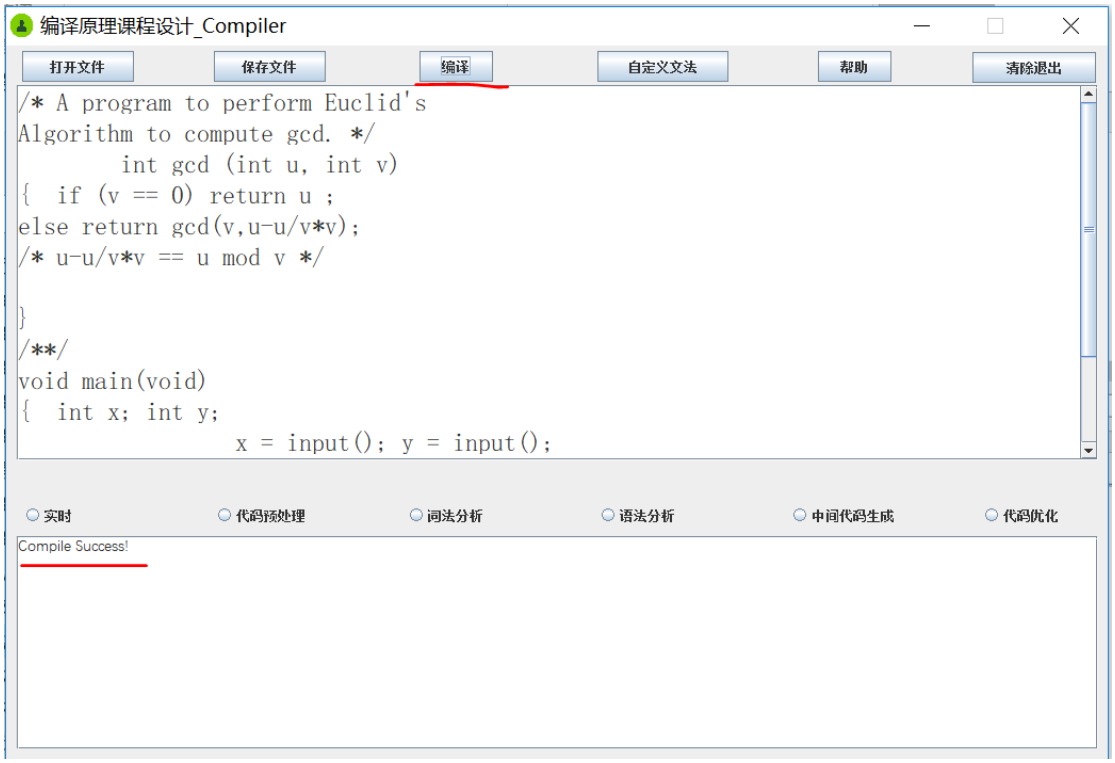
[图]4.1(2)打开文件

(2)保存*.c 文件



[图]4.1(3)保存文件

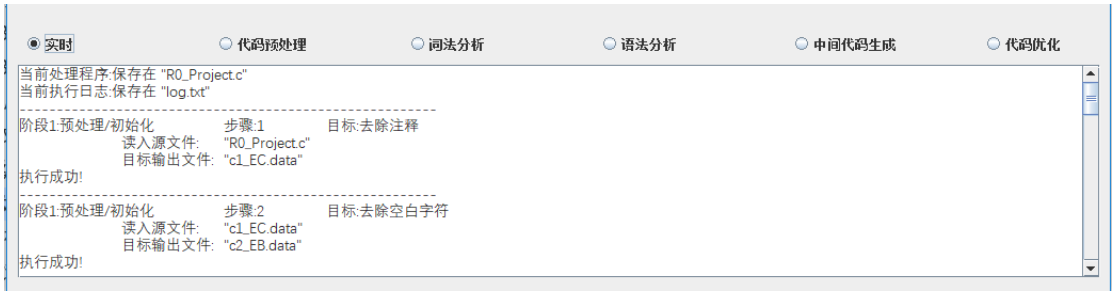
4.2 编译



[图]4.2 编译

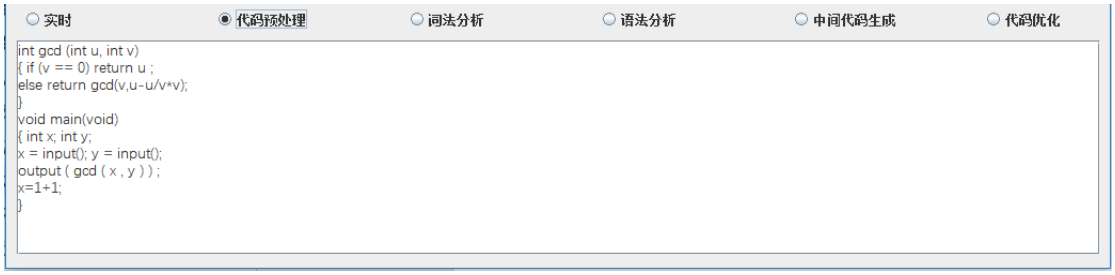
4.3 处理步骤

(1)实时



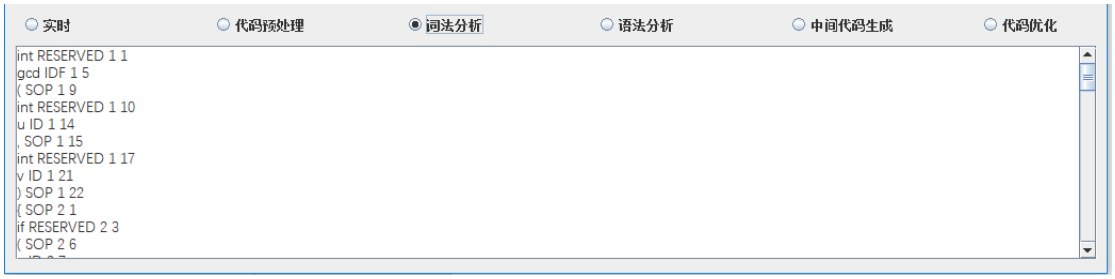
[图]4.3(1)实时

(2)代码预处理



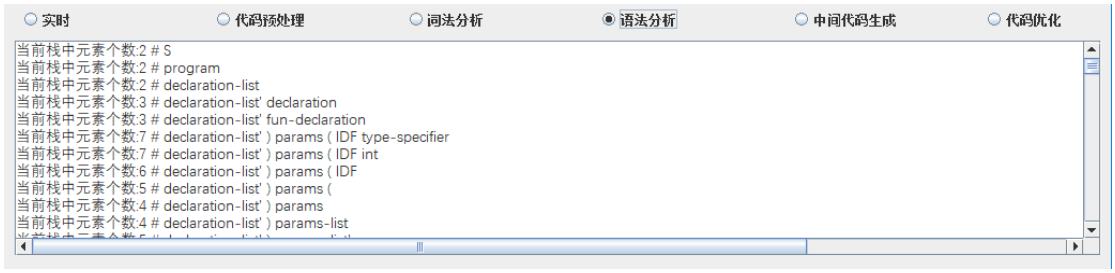
[图]4.3(2)代码预处理

(3)词法分析



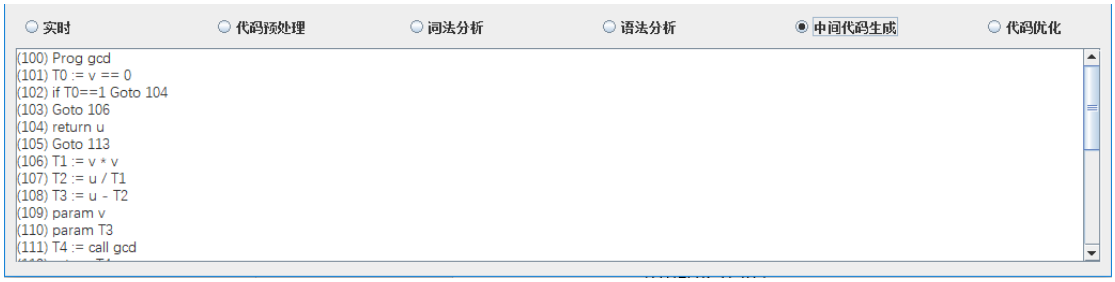
[图]4.3(3)词法分析

(4)语法分析



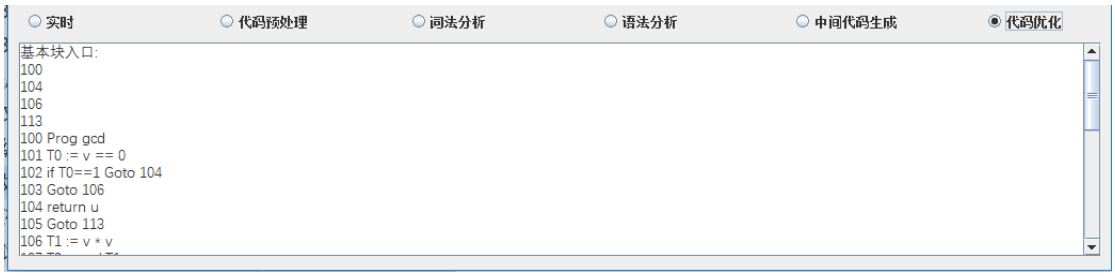
[图]4.3(4)语法分析

(5)中间代码生成



[图]4.3(5)中间代码生成

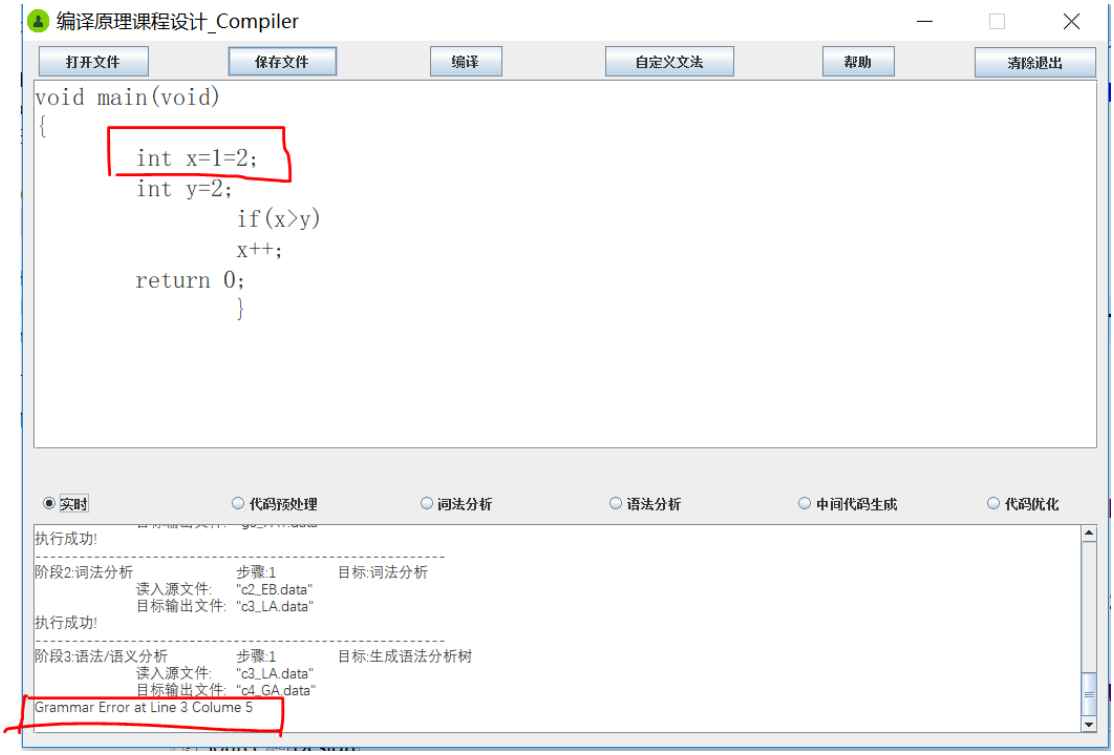
(6)代码优化



[图]4.3(6)代码优化

4.4 错误处理

错误测试用例代码如下图，第三行代码”int x=1=2;”不符合C语言语法规则。点击编译按钮后，在实时控制台显示“Grammer Error at Line 3 Colume 5”错误提示。



[图]4.4 错误处理

五、核心源码

5.1 词法分析

```
//词法分析
void lexicalanalysis(fstream &in, fstream &out)
{
	string strToken;
	char ch;
	while (!in.eof())
	{
		ch = in.get();
		if (in.fail())
			break;

		if (isBC(ch)) { //遇到空白字符
			strToken = "";
		}
		else if (isalpha(ch)) { //遇到字母

			while (isalpha(ch) || isdigit(ch)) { //读取整个token  直到下一个空白字符
				strToken.push_back(ch);
				column++;
				ch = in.get();
			}

			if (Reserve(strToken)) { //判断token是否为保留字
				//cout << strToken << ", RESERVED" << endl;
				out << strToken << " RESERVED" << " " << line << " " << column -
strToken.length() << endl;
			}
			else {
				// cout << strToken << ", ID" << endl;
				out << strToken << " ID" << " " << line << " " << column - strToken.length()
<< endl;
			}

			strToken = "";
			Retract(in);
		}
		else if (isdigit(ch)) { //遇到数字
			while (isdigit(ch)) {//读取整个数
				strToken.push_back(ch);
				column++;
```

```
                ch = in.get();
            }
            //数字之后出现字母 发现错误格式的ID 报错
            if (isalpha(ch)) {
                cout << "[Lexical ERROR] " << " [" << line << "," << column -
strToken.length() << "] " << "Invalid ID: ";

                while (isalpha(ch) || isdigit(ch)) {
                    strToken.push_back(ch);
                    column++;
                    ch = in.get();
                }

                cout << "\"" << strToken << "\"" << endl;
                out << "^ " << strToken << " ID " << line << " " << column -
strToken.length() << endl;
                out << "[Lexical ERROR] " << " [" << line << "," << column -
strToken.length() << "] " << "Invalid ID: ";
                out << "\"" << strToken << "\"" << endl;
            }
            else {
                //cout << strToken << ", INT" << endl;
                out << strToken << " NUM" << " " << line << " " << column -
strToken.length() << endl;
            }

            Retract(in);
            strToken = "";
        }
        else {
            switch (ch) //其他OP类字符
            {
            case '=':
                column++;
                ch = in.get();
                if (ch == '=')
                {
                    column++;
                    out << "== COP" << " " << line << " " << column - 2 << endl;
                }
                else {
                    out << "=    AOP" << " " << line << " " << column - 1 << endl;
                    Retract(in);
                }
```

```
                break;
            case '<':
                column++;
                ch = in.get();
                if (ch == '=') {
                    column++;
                    //cout << "<=, COP" << endl;
                    out << "<= COP" << " " << line << " " << column - 2 << endl;
                }
                else {
                    //cout << "<, COP" << endl;
                    out << "<   COP" << " " << line << " " << column - 1 << endl;
                    Retract(in);
                }
                break;
            case '>':
                column++;
                ch = in.get();
                if (ch == '=') {
                    column++;
                    //cout << ">=, COP" << endl;
                    out << ">=   COP" << " " << line << " " << column - 2 << endl;
                }
                else {
                    //cout << ">, COP" << endl;
                    out << ">   COP" << " " << line << " " << column - 1 << endl;
                    Retract(in);
                }
                break;
            case '!':
                column++;
                ch = in.get();
                if (ch == '=') {
                    column++;
                    out << "!=   COP" << " " << line << " " << column - 2 << endl;
                }
                else
                {
                    //cout << "[Lexical ERROR] " << " [" << line << "," << column << "] "
<< "Invalid COP ";
                    out << "[Lexical ERROR] " << " [" << line << "," << column - 1 << "] "
<< "Invalid COP ";
                    Retract(in);
                }
```

```
                    break;
                case '+':
                case '-':
                case '/':
                case '*':
                    column++;
                    //cout << ch << ", OOP" << endl;
                    out << ch << " OOP" << " " << line << " " << column - 1 << endl;
                    break;
                case ';':
                    column++;
                    //cout << ch << ", EOP" << endl;
                    out << ch << " EOP" << " " << line << " " << column - 1 << endl;
                    break;
                case '(':
                case ')':
                case ',':
                case '[':
                case ']':
                case '{':
                case '}':
                    column++;
                    //cout << ch << ", SOP" << endl;
                    out << ch << " SOP" << " " << line << " " << column - 1 << endl;
                    break;
                default:   //未识别成功的信息  显示出错信息
                    column++;
                    //cout << ch << ", UNKNOWN" << endl;
                    out << "[Lexical ERROR] " << " [" << line << "," << column - 1 << "] " <<
"UNKNOW EXPRESSION " << ch << endl;
                }
            }
        }
        out << "# # " << line + 1 << " 1";
}
```

5.2 语法分析

```
//计算终结符集合以及非终结符集合
void cal_vn_vt(fstream &in, fstream &out)
{
    string temp, part1, part2, t;
    int split, start, index, num = 1;
```

```
    init_vttype(vttype);
    while (!in.eof())
    {
        getline(in, temp);
        if (in.fail())
            break;
        temp.replace(0, temp.find(' '), "");  //去掉每行开头的数字
        split = temp.find("->");
        part1.assign(temp, 0, split);
        part2.assign(temp, split + 2, temp.length() - split - 1);
        fix(part1);
        fix(part2);                    //将每一行语法分为箭头前后的两个部分
        start = 0;
        grammarmap.insert({ num,{part1} });
        //左侧部分  如果vn集合中还没有出现则加入  如果vt集合中出现则将vt集合中的该元素删除
        if (!check_exist(vn, part1))
            vn.push_back(part1);
        if (check_exist(vt, part1))
            del(vt, part1);
        while (1)        //将箭头后的部分按照空格分开
        {
            index = part2.find(' ', start);
            if (index == part2.npos)
                break;
            t.assign(part2, start, index - start);
            start = index + 1;
            //当右侧部分在  vn中没有出现且vt中没有出现  则将该元素加入vt集合中
            if (!check_exist(vn, t) && !check_exist(vt, t))
                vt.push_back(t);
            grammarmap[num].push_back(t);
        }
        t.assign(part2, start, part2.length() - start);
        if (!check_exist(vn, t) && !check_exist(vt, t))
            vt.push_back(t);
        //填充可以推导出空的非终结符表
        grammarmap[num].push_back(t);
        if (grammarmap[num][1] == "empty")
            emptymake.insert({ grammarmap[num][0],num });
        num++;
    }
    vt.push_back("#");
    //输出
    output_vec(out);
```

```
}

//输出vector中所有信息  输出vn,vt信息至文件
void output_vec(fstream &out)
{
	out << "非终结符vn:" << endl;
	for (iter = vn.begin(); iter != vn.end(); iter++)
	{
		out << *iter << endl;
		firstlist.insert({ *iter, {} });
		followlist.insert({ *iter,{} });
		if (*iter == firststr)
			followlist[firststr].push_back("#");
	}
	out << "终结符vt:" << endl;
	for (iter = vt.begin(); iter != vt.end(); iter++)
	{
		if (!checkmap(vttype, *iter))
		{
			//cout << "Undefined vt[ " << *iter << " ]!" << endl;
			//exit(0);
			;
		}
		out << *iter << endl;
		firstlist.insert({ *iter, {*iter} });
	}
}

//求first集
void cal_first()
{
	int flag, outflag;
	unsigned int k;
	string vnname, tname, temp;
	while (1)
	{
		flag = 1;
		for (gmaper = grammarmap.begin(); gmaper != grammarmap.end(); gmaper++)
		{
			vnname = gmaper->second[0];
			if (check_exist(vt, gmaper->second[1]))
			{
				if (!check_exist(firstlist[vnname], gmaper->second[1]))
				{
```

```
                    firstlist[vnname].push_back(gmaper->second[1]);
                    flag = 0;
                }   //判断该终结符是否已经在该推导的first集中,没有则添加
            }       //产生式第一个元素为终结符
            else
            {
                k = 1;
                outflag = 1;
                while (1)
                {
                    if (k >= gmaper->second.size())
                        break;
                    //cout << k << " " << gmaper->second.size() << endl;
                    tname = gmaper->second[k];
                    if (check_exist(vn, tname))
                    {
                        //逐个检验是否已经在vnname的first集中,不存在就添加
                        for (iter = firstlist[tname].begin(); iter != firstlist[tname].end();
iter++)
                        {
                            temp = *iter;
                            if (!check_exist(firstlist[vnname], temp))
                            {
                                firstlist[vnname].push_back(temp);
                                flag = 0;
                            }
                            if (*iter == "empty")      //如果该非终结符的first集中没有
empty则进入下一推导式
                                outflag = 0;
                        }
                    }
                    else
                    {       //推导式中出现终结符 判断是否添加后 进入下一个推导
式
                        outflag = 1;
                        if (!check_exist(firstlist[vnname], tname))
                        {
                            firstlist[vnname].push_back(tname);
                            flag = 0;
                        }
                    }
                    if (outflag)
                        break;
                    k++;
```

```
                    }
                }
            }
            //当flag为1时表示已经没有first集发生添加元素  fisrt集合求解完成
            if (flag)
                break;
        }
}

//求follow集
void cal_follow()
{
    int flag, outflag, exflag;
    string vnname, tname;
    vector<string>::iterator titer, temp;
    while (1)
    {
        flag = 1;
        for (gmaper = grammarmap.begin(); gmaper != grammarmap.end(); gmaper++)       //
遍历推导式
        {
            for (iter = gmaper->second.begin() + 1; iter != gmaper->second.end(); iter = titer)
            //遍历推导式中个符号
            {
                titer = iter + 1;
                if (check_exist(vt, *iter))             //遇到终结符  不需要计算follow集
                    continue;
                else
                {
                    vnname = *iter;             //遇到一个非终结符开始向后遍历  知道遇到
不能推导空的非终结符或是终结符
                    while (1)
                    {
                        iter++;
                        outflag = 1;            //记录一个非终结符能不能推导出空
                        if (iter == gmaper->second.end())       //到推导式结束  当前非终
结符follow集操作完成
                            break;
                        if (check_exist(vt, *iter))
                        {
                            if (!check_exist(followlist[vnname], *iter) && *iter !=
"empty")
                            {
                                followlist[vnname].push_back(*iter);
```

```
                                flag = 0;
                            }
                        }       //遇到一个终结符 判断是否存在 不存在则加入当前非终结符的follow集中
                        else
                        {
                            for (temp = firstlist[*iter].begin(); temp != firstlist[*iter].end(); temp++)
                            {
                                if (!check_exist(followlist[vnname], *temp) && *temp != "empty")
                                {
                                    followlist[vnname].push_back(*temp);
                                    flag = 0;
                                }
                                if (*temp == "empty")
                                    outflag = 0;
                            }       //将当前符号后遇到的所有的first集合判断并加入到follow集中 直到遇到不能推导出空的非终结符
                        }
                        if (outflag)
                            break;
                    }
                }
            }
            unsigned int i = 1;
            //推导式最后连续i个非终结符可以推导为空 则推导式最前的非终结符的follow集中所有元素判断并添加到这所有的非终结符的follow集中
            while (1)
            {
                if (check_exist(vn, *(gmaper->second.end() - i)))
                {
                    string str;
                    if (i >= gmaper->second.size())
                        break;
                    str = *(gmaper->second.end() - i);
                    vnname = gmaper->second[0];
                    for (temp = followlist[vnname].begin(); temp != followlist[vnname].end(); temp++)
                    {
                        if (!check_exist(followlist[str], *temp) && *temp != "empty")
                        {
                            followlist[str].push_back(*temp);
                            flag = 0;
```

```
                        }
                    }
                    exflag = 0;
                    for (iter = firstlist[str].begin(); iter != firstlist[str].end(); iter++)
                        if (*iter == "empty")
                            exflag = 1;
                    //exflag记录该非终结符的first集中是否有empty
                    if (!exflag)
                        break;
                    else
                        i++;
                }
                else
                    break;
            }
        }
        if (flag)
            break;
    }
}

//计算first follow集
void cal_first_follow(fstream &in, fstream &out)
{
    cal_first();
    cal_follow();
    output_ff(out);
}

//输出first,follow集至文件
void output_ff(fstream &out)
{
    //输出first集
    out << "//first集" << endl;
    for (maper = firstlist.begin(); maper != firstlist.end(); maper++)
    {
        if (!check_exist(vt, maper->first))
        {
            out << maper->first << "的first集:";
            for (iter = maper->second.begin(); iter != maper->second.end(); iter++)
                out << *iter << " ";
            out << endl;
        }
    }
```

```
    //输出follow集
    out << "//follow集" << endl;
    for (maper = followlist.begin(); maper != followlist.end(); maper++)
    {
        if (!check_exist(vt, maper->first))
        {
            out << maper->first << "的follow集:";
            for (iter = maper->second.begin(); iter != maper->second.end(); iter++)
                out << *iter << " ";
            out << endl;
        }
    }
}
```

5.3 语义分析

```
//生成语法分析树
void make_tree(fstream &in, fstream &out)
{
    string str;
    int num, line;
    struct Gtree *cur;
    tree_init();
    stack_init(stack);                    //初始化语法分析树以及语法栈
    cur = treenode->sons[2];
    while (!in.eof())
    {
        if (!stack.current)
            break;
        in >> skipws;
        in >> token.re >> token.type >> token.line >> token.colume;
        if (in.fail())
            break;                        //从词法分析结果中逐个读取token
        stack_show(stack, out);
        str = stack_gettop(stack);
        while (mtable.vtname.find(str) == mtable.vtname.end())
        {   //直到栈顶元素为终结符 跳出循环 准备比较\出栈操作
            if (token.type != "NUM" && token.type != "ID" && token.type != "IDF" &&
token.type != "ID1")
                num = mtable.vtname[token.re];
            else
                num = mtable.vtname[token.type];      //数字 ID类型token通过原始字符串
得到编号 其余通过type类型
            if (mtable.mtable[mtable.vnname[str]][num] == 0)
```

```
                {
                    out << "Grammar Error at Line " << token.line << " Colume " <<
token.colume << endl;
                    cout << "Grammar Error at Line " << token.line << " Colume " <<
token.colume << endl;
                    //cout << str << endl;
                    //cout << token.re << endl;
                    exit(0);
                }           //预测分析表行列所对应的推导式标号为0  表示出现语法错误
                else
                {
                    line = mtable.mtable[mtable.vnname[str]][num];
                    stack_popout(stack);
                    cur->bitnum = 0;
                    cur->mtnum = line;
                    for (iter = grammarmap[line].end() - 1; iter != grammarmap[line].begin(); iter-
-)
                    {
                        if (*iter != "empty")
                        {
                            stack_pushin(stack, *iter);
                        }
                    }           //将不为empty的产生式  压入栈中

                    for (iter = grammarmap[line].begin() + 1; iter != grammarmap[line].end();
iter++)
                    {
                        if (cur->mtnum == 42)
                        {
                            if (*iter == "expression")
                            {
                                cur->bitnum++;
                                cur->sons[cur->bitnum] = (treebit *)malloc(sizeof(treebit));
                                cur->sons[cur->bitnum - 1]->brother =
cur->sons[cur->bitnum];
                                cur->sons[cur->bitnum]->father = cur;
                                strcpy_s(cur->sons[cur->bitnum]->re, "$42_1");
                                strcpy_s(cur->sons[cur->bitnum]->type, "todo");
                            }
                            if (*iter == ")")
                            {
                                cur->bitnum++;
                                cur->sons[cur->bitnum] = (treebit *)malloc(sizeof(treebit));
                                cur->sons[cur->bitnum - 1]->brother =
```

```
cur->sons[cur->bitnum];
                    cur->sons[cur->bitnum]->father = cur;
                    strcpy_s(cur->sons[cur->bitnum]->re, "$42_2");
                    strcpy_s(cur->sons[cur->bitnum]->type, "todo");
                }
                //42 iteration-stmt -> while ( $42_1 expression $42_2 ) statement
$42_3
            }       //while语句推导中需要增加两个中间语义动作
            if (cur->mtnum == 39)
            {
                if (*iter == ")")
                {
                    cur->bitnum++;
                    cur->sons[cur->bitnum] = (treebit *)malloc(sizeof(treebit));
                    cur->sons[cur->bitnum - 1]->brother =
cur->sons[cur->bitnum];
                    cur->sons[cur->bitnum]->father = cur;
                    strcpy_s(cur->sons[cur->bitnum]->re, "$39_1");
                    strcpy_s(cur->sons[cur->bitnum]->type, "todo");
                }
                if (*iter == "selection-stmt'")
                {
                    cur->bitnum++;
                    cur->sons[cur->bitnum] = (treebit *)malloc(sizeof(treebit));
                    cur->sons[cur->bitnum - 1]->brother =
cur->sons[cur->bitnum];
                    cur->sons[cur->bitnum]->father = cur;
                    strcpy_s(cur->sons[cur->bitnum]->re, "$39_2");
                    strcpy_s(cur->sons[cur->bitnum]->type, "todo");
                }   //39 selection-stmt -> if ( expression $39_1 ) statement $39_2
selection-stmt' $39_3
            }       //if语句推导中需要增加两个中间语义动作
            if (cur->mtnum == 41)
            {
                if (*iter == "statement")
                {
                    cur->bitnum++;
                    cur->sons[cur->bitnum] = (treebit *)malloc(sizeof(treebit));
                    cur->sons[cur->bitnum - 1]->brother =
cur->sons[cur->bitnum];
                    cur->sons[cur->bitnum]->father = cur;
                    strcpy_s(cur->sons[cur->bitnum]->re, "$41_1");
                    strcpy_s(cur->sons[cur->bitnum]->type, "todo");
                }   //41 selection-stmt' -> else $41_1 statement $41_2
```

```
                }           //else语句推导中需要增加一个中间语义动作
                cur->bitnum++;
                cur->sons[cur->bitnum] = (treebit *)malloc(sizeof(treebit));
                if (cur->bitnum > 1)
                    cur->sons[cur->bitnum - 1]->brother = cur->sons[cur->bitnum];
                cur->sons[cur->bitnum]->father = cur;
                strcpy_s(cur->sons[cur->bitnum]->re, (*iter).c_str());
                if (mtable.vnname.find(*iter) != mtable.vnname.end())
                    strcpy_s(cur->sons[cur->bitnum]->type, "derivation");
                else
                    strcpy_s(cur->sons[cur->bitnum]->type,
vttype.find(*iter)->second.c_str());
                cur->sons[cur->bitnum]->info = (struct info *)malloc(sizeof(struct info));
                nodenum++;
                cur->sons[cur->bitnum]->nodenum = nodenum;
                placelistmap.insert({ nodenum,{} });
            }           //将该条产生式的所有元素 加入到当前节点的孩子节点 并填
充相关信息以及兄弟父亲节点
            cur->bitnum++;
            stringstream ss;
            ss << cur->mtnum;
            cur->sons[cur->bitnum] = (treebit *)malloc(sizeof(treebit));
            strcpy_s(cur->sons[cur->bitnum]->re, ("$" + ss.str()).c_str());
            if (ss.str() == "42")
                strcpy_s(cur->sons[cur->bitnum]->re, "$42_3");
            if (ss.str() == "39")
                strcpy_s(cur->sons[cur->bitnum]->re, "$39_3");
            if (ss.str() == "41")
                strcpy_s(cur->sons[cur->bitnum]->re, "$41_2");
            //语义分析节点命名由 $ 符号 加上 产生式编号组成
            strcpy_s(cur->sons[cur->bitnum]->type, "todo");
            cur->sons[cur->bitnum]->father = cur;
            cur->sons[cur->bitnum - 1]->brother = cur->sons[cur->bitnum];
            cur->sons[cur->bitnum]->brother = NULL;             //最后添加$节点 表
明用第几条产生式产生 便于后续语义分析
            cur = cur->sons[1];
            if (strcmp(cur->re, "empty") == 0)
            {
                while (1)
                {
                    while (!cur->brother)
                    {
                        cur = cur->father;
                        if (strcmp(cur->re, "head") == 0)
```

```
                                    break;
                                }
                                cur = cur->brother;
                                if (cur->re[0] != '$')
                                    break;
                            }
                        }          //应用完当前产生式 进入兄弟节点继续推导 没有兄弟节点 返回父亲节点 进入它的兄弟节点 知道兄弟节点存在
                            str = stack_gettop(stack);
                            stack_show(stack, out);
                        }
                    }
                    if (token.re != stack_gettop(stack) && token.type != stack_gettop(stack))
                    {
                        out << "Grammar Error at Line " << token.line << " Colume " << token.colume << endl;
                        cout << "Grammar Error at Line " << token.line << " Colume " << token.colume << endl;
                        exit(0);
                    }                   //栈顶终结符类型与当前token类型不符合 报错
                    if (strstr(cur->type, "ID"))
                        strcpy_s(cur->re, token.re.c_str());       //对于id类型直接改变当前节点原始字符串 不再产生新的节点
                    if (strstr(cur->type, "NUM"))
                        strcpy_s(cur->re, token.re.c_str());
                    while (1)
                    {
                        while (!cur->brother)
                        {
                            cur = cur->father;
                            if (strcmp(cur->re, "head") == 0)
                                break;
                        }
                        if (strcmp(cur->re, "head") == 0)
                            break;
                        cur = cur->brother;
                        if (cur->re[0] != '$')
                            break;
                    }          //寻找满足条件的兄弟节点
                    //栈顶元素出栈
                    stack_popout(stack);
                }
            }
```

5.4 中间代码生成与优化

```
//读入所有三地址代码
void getbetter1(fstream &in) {
	string str, part1, part2, t;
	int split = 0, index, start, num;
	while (!in.eof()) {
		getline(in, str);
		if (in.fail())
			break;
		split = str.find(' ');
		part1.assign(str, 0, split);
		part1 = part1.substr(1, part1.length() - 1);
		part2.assign(str, split + 1, str.length() - split - 1);
		fix(part1);
		fix(part2);
		//读入每一行中间代码 将代码地址 与代码内容分开
		start = 0;
		stringstream ss;
		ss << part1;
		num = atoi(ss.str().c_str());
		gb.insert({ num,{} });
		//将地址提取为数字 插入代码表中
		while (1) {
			index = part2.find(' ', start);
			if (index == part2.npos)
				break;
			t.assign(part2, start, index - start);
			start = index + 1;
			gb[num].push_back(t);
		}
		t.assign(part2, start, part2.length() - start);
		gb[num].push_back(t);
	}
	//得到所有基本块的入口
	for (gber = gb.begin(); gber != gb.end(); gber++)
	{
		//1.程序的第一条语句
		if (gber == gb.begin())
			entry.push_back(gber->first);
		//2.强制跳转语句的目的地址
		for (ier = gber->second.begin(); ier != gber->second.end(); ier++) {
			if (*ier == "Goto") {
				iertemp = ier + 1;
```

```
                    stringstream ss;
                    ss << *iertemp;
                    num = atoi(ss.str().c_str());
                    itertemp = find(entry.begin(), entry.end(), num);
                    if (itertemp == entry.end())
                        entry.push_back(num);
                }
                //3.条件语句的下一条语句
                if (*ier == "if") {
                    gbertemp = gber;
                    gbertemp++;
                    gbertemp++;
                    itertemp = find(entry.begin(), entry.end(), gbertemp->first);
                    if (itertemp == entry.end())
                        entry.push_back(gbertemp->first);
                }
                //4.函数的第一条语句
                if (*ier == "Proc")
                    entry.push_back(gber->first);
            }
        }
        gber--;
        //记录所有代码地址的最后一条
        endline = gber->first;
}

//
int checkdigit(string str)
{
    int i;
    for (i = 0; i < str.length(); i++)
        if (!isdigit(str[i]))
            return 0;
    return 1;
}

//
int checkvar(string str, map<string, double> cons)
{
    for (con = cons.begin(); con != cons.end(); con++)
        if (con->first == str)
            return 1;
    return 0;
}
```

```
//
void getbetter2(int start, int end)
{
	int i;
	int flag = 0;
	map<string, double> constant;
	for (i = start; i <= end; i++)
	{
		flag = 0;
		if (gb[i].size() == 3 && checkdigit(gb[i][2]))
		{
			stringstream ss;
			ss << gb[i][2];
			double ll;
			ll = atol(ss.str().c_str());
			constant.insert({ gb[i][0] ,ll });
		}
		if (gb[i].size() == 3)
		{
			for (con = constant.begin(); con != constant.end(); con++)
			{
				if (con->first == gb[i][2])
					gb[i][2] = to_string(con->second);
			}
		}
		if (gb[i].size() == 5)
		{
			if (checkdigit(gb[i][2]))
				flag++;
			if (checkdigit(gb[i][4]))
				flag++;
			for (con = constant.begin(); con != constant.end(); con++)
			{
				if (con->first == gb[i][2])
				{
					gb[i][2] = to_string(con->second);
					flag++;
				}
				if (con->first == gb[i][4])
				{
					gb[i][4] = to_string(con->second);
					flag++;
				}
```

```
}
if (flag == 2)
{
	double b, d;
	b = atol(gb[i][2].c_str());
	d = atol(gb[i][4].c_str());
	switch (op[gb[i][3]])
	{
	case 1:
		gb[i][2] = to_string(b + d);
		break;
	case 2:
		gb[i][2] = to_string(b - d);
		break;
	case 3:
		gb[i][2] = to_string(b * d);
		break;
	case 4:
		gb[i][2] = to_string(b / d);
		break;
	case 5:
		gb[i][2] = to_string(b == d);
		break;
	case 6:
		gb[i][2] = to_string(b != d);
		break;
	case 7:
		gb[i][2] = to_string(b <= d);
		break;
	case 8:
		gb[i][2] = to_string(b < d);
		break;
	case 9:
		gb[i][2] = to_string(b >= d);
		break;
	case 10:
		gb[i][2] = to_string(b > d);
		break;
	};
	gb[i].pop_back();
	gb[i].pop_back();
	constant.insert({ gb[i][0],atol(gb[i][2].c_str()) });
}
if (checkvar(gb[i][0], constant) && flag == 0)
```

```
                {
                    constant.erase(gb[i][0]);
                }
            }
        }
}

//优化中间代码
void optimize(fstream &in, fstream &out)
{
    init_op();
    getbetter1(in);
    //输出基本块信息
    out << "基本块入口:" << endl;
    for (it = entry.begin(); it != entry.end(); it++)
        out << *it << endl;
    for (it = entry.begin(); it != entry.end(); it++)
    {
        itertemp = it;
        itertemp++;
        if (itertemp == entry.end())
            getbetter2(*it, endline);
        else
            getbetter2(*it, *itertemp - 1);
    }
    for (gber = gb.begin(); gber != gb.end(); gber++)
    {
        out << gber->first << " ";
        for (ier = gber->second.begin(); ier != gber->second.end(); ier++)
            out << *ier << " ";
        out << endl;
    }
}
```

六、总结

七、参考文献