目录

一、项目概况

1.1 目的

1. 掌握使用高级程序语言实现一个一遍完成的、简单语言的编译器的方法。
2. 掌握简单的词法分析器、语法分析器、符号表管理、中间代码生成以及目标代码生成的实现方法。
3. 掌握将生成代码写入文件的技术

1.2 要求

使用高级程序语言实现一个类 C 语言的编译器，可以提供词法分析、语法分析、符号表管理、中间代码生成以及目标代码生成等功能。具体要求如下：

（1） 使用高级程序语言作为实现语言，实现一个类 C 语言的编译器。编码实现编译器的组成部分。

（2） 要求的类 C 编译器是个一遍的编译程序，词法分析程序作为子程序，需要的时候被语法分析程序调用。

（3） 使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。

（4） 要求输入类 C 语言源程序，输出中间代码表示的程序;

（5） 要求输入类 C 语言源程序，输出目标代码（可汇编执行）的程序。

（6） 实现过程、函数调用的代码编译。

（7） 拓展类 C 语言文法，实现包含数组的中间代码以及目标代码生成。

其中（1)、(2)(3)(4)(5）是必做内容，(6)(7）是选作内容。

1.3 完成概况

我完成了包括选作内容在内的 (1)-(7) 全部内容。

选择 C++作为编程语言，使用 Qt 平台，以图形化界面实现编译过程的展示,展示部分包括词法分析时识别的终结符，语法分析时的 First、Follow 集、SLR(1)分析表，语义分析时生成的中间代码，代码生成时的目标代码。

程序能对常见错误进行处理提示。

1.4 开发环境

操作系统：Windows 10 19042.1237
硬件环境：Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz
Qt 环境：Desktop Qt 5.12.10 MinGW 64-bit Debug

二、需求分析

2.1 输入

类 C 语言代码，代码中必须包括 main 函数

2.2 输出

在 QT 图形化界面上显示的内容：
1. 词法分析得到的终结符
2. 语法分析得到的 First 集、Follow 集、SLR(1)分析表
3. 语义分析得到的中间代码（四元式形式）
4. 代码生成得到的 MIPS 指令

写入文件的内容
1. 语义分析得到的中间代码（四元式形式）
2. 代码生成得到的 MIPS 指令

2.3 错误处理

序号	可能出现的错误	处理方式
1	打开源程序/产生式文件失败	对话框弹出对应提示
2	打开要输入中间代码/目标代码的文件失败	在 Qt 界面中对应输出中间代码/目标代码的文本框中输出对应提示
3	词法分析时遇到未知符号	在输出终结符文本框中输出对应提示错误字符，弹窗提示
4	语法分析失败	在输出语法分析的文本框中输出对应提示
5	语义分析中遇到不符合语法规则的情况	在输出中间代码的文本框中输出对应提示
6	使用未声明变量/函数	在输出中间代码的文本框中输出对应提示
7	缺少 main 函数	在输出中间代码的文本框中输出对应提示
8	形参与实参个数不匹配	在输出中间代码的文本框中输出对应提示
9	未进行中间代码生成过程而直接生成目标代码	在输出目标代码的文本框中输出对应提示

2.4 程序功能

针对 2.1 中的输入，依次进行词法分析、语法分析、语义分析及中间代码生成、代

码优化、目标代码生成，得到 2.2 中的输出。

程序可识别的单词及语法见下图。

程序能对 2.3 中的错误进行一定处理。

可识别单词

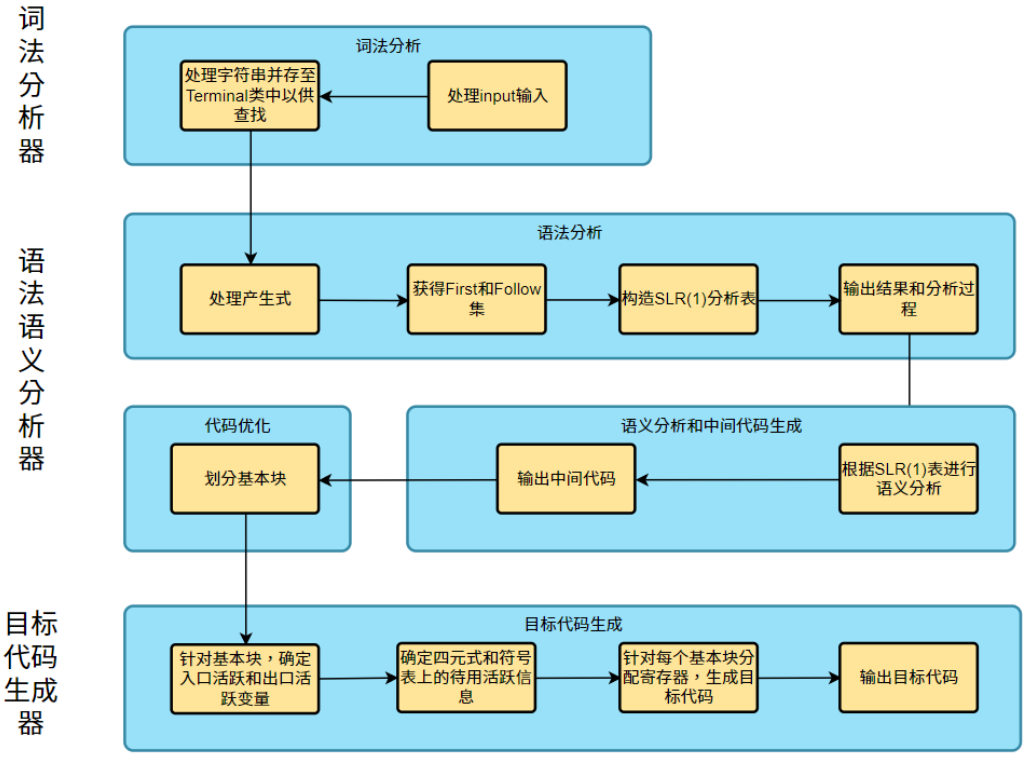
关键字：int | void | if | else | while | return
标识符： 字母（字母|数字）* （注：不与关键字相同）
数值：数字（数字）*
赋值号： =
算符： + | - | * | / | = | == | > | >= | < | <= | !=
界符： ；
分隔符： ，
注释号： /* */ | //
左括号： （
右括号： ）
左大括号： {
右大括号： }
字母：| a |....| z | A |....| Z |
数字：0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
结束符：#

可识别语法

Program ::= <声明串>
<声明串> ::=<声明>{<声明>}
<声明> ::=int <ID> <声明类型> | void <ID> <函数声明>
<声明类型>::=<变量声明> | <函数声明> | <数组声明>
<变量声明> ::= ;
<函数声明> ::= ‘(’<形参>‘)’ <语句块>
<数组声明> ::= ‘[’ (数字)$^+$ ‘]’{‘[’ (数字)$^+$ ‘]’}
<形参>::= <参数列表> | void
<参数列表> ::= <参数> {, <参数>}
<参数> ::= int <ID>
<语句块> ::= ‘{’<内部声明> <语句串>‘}’
<内部声明> ::= 空 |<内部变量声明>;{<内部变量声明>;}
<内部变量声明>::=int <ID>
<语句串> ::= <语句>{ <语句> }
<语句> ::= <if 语句> |< while 语句> | <return 语句> | <赋值语句>
<赋值语句> ::= <ID>‘=’<表达式> | <数组>‘=’<表达式>;
<return 语句> ::= return [<表达式>] ;**（注：[ ]中的项表示可选）**
<while 语句> ::= while ‘(’<表达式> ‘)’ <语句块>

三、系统设计

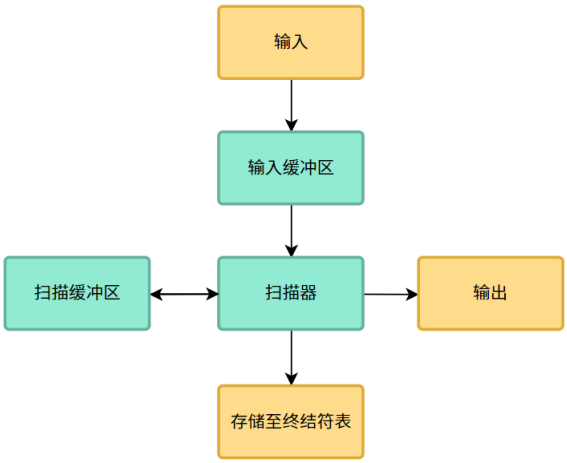
编译器主要分为词法分析、语法分析、语义分析、代码优化、目标代码生成五部分。出于设计的简便性考虑，将语法分析和语义分析合为一个语法语义分析器。代码优化部分划分了基本块，由于该部分比较简单，也一并合入语法语义分析器。针对词法分析和目标代码生成分别设计了词法分析器和目标代码生成器。



系统总设计流程图

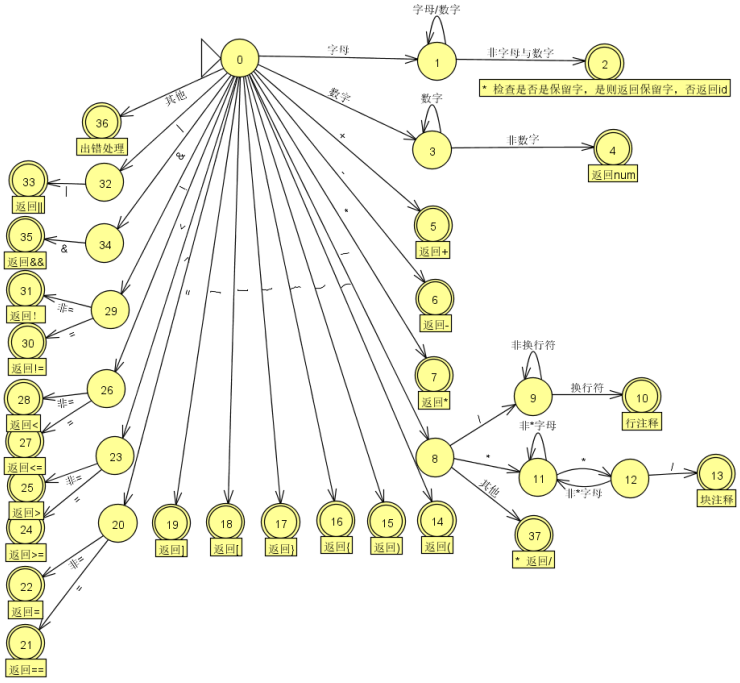
3.1 词法分析器

3.1.1 词法分析器结构



词法分析器结构图

3.1.2 单词识别状态转换图



单词识别状态转换图

3.2 语法和语义分析器

3.2.1 语法分析使用的产生式

因需要使用语法制导的翻译技术，所以对 C 语言文法规则的产生式进行改进。改进后的语法产生式如下：

```
P ::= N declare_list
declare_list ::= declare declare_list
declare_list ::= declare
declare ::= int ID M A func_declare
declare ::= int ID ;
declare ::= void ID M A func_declare
A ::=
func_declare ::= ( para ) stc_block
para ::= para_list
para ::= void
para_list ::= param
para_list ::= param , para_list
param ::= int ID
stc_block ::= { inner_declare stc_list }
inner_declare ::=
```

```
inner_declare ::= inner_var_declare ; inner_declare
inner_var_declare ::= int ID
stc_list ::= stc M stc_list
stc_list ::= stc
stc ::= if_stc
stc ::= while_stc
stc ::= return_stc
stc ::= assign_stc
assign_stc ::= ID = exp ;
return_stc ::= return ;
return_stc ::= return exp ;
while_stc ::= while M ( exp ) A stc_block
if_stc ::= if ( exp ) A stc_block
if_stc ::= if ( exp ) A stc_block N else M A stc_block
N ::=
M ::=
exp ::= add_exp
exp ::= add_exp > add_exp
exp ::= add_exp < add_exp
exp ::= add_exp == add_exp
exp ::= add_exp >= add_exp
exp ::= add_exp <= add_exp
exp ::= add_exp != add_exp
add_exp ::= item
add_exp ::= item + add_exp
add_exp ::= item - add_exp
item ::= factor
item ::= factor * item
item ::= factor / item
factor ::= NUM
factor ::= ( exp )
factor ::= ID ( argu_list )
factor ::= ID
argu_list ::=
argu_list ::= exp
argu_list ::= exp , argu_list
factor ::= array
array ::= ID [ factor ]
array ::= ID [ factor ] [ factor ]
inner_var_declare ::= int ID [ factor ]
inner_var_declare ::= int ID [ factor ] [ factor ]
assign_stc ::= array = exp ;
```

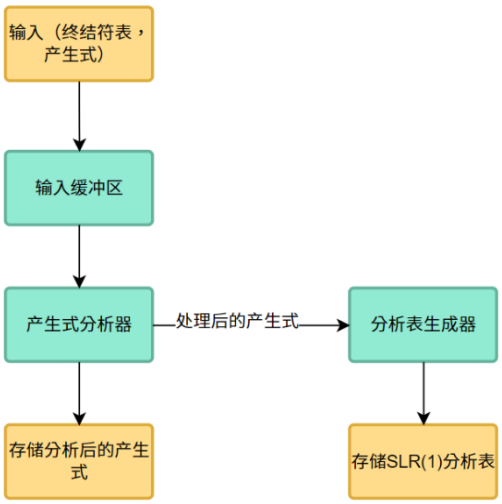
production.conf 语法规则

非终结符	助记符
<源程序>	P
A（记录语句块的层次）	A
M（记录某些位置的四元式标号）	M
N（以生成一条跳转 指令和一个链）	N
<ID>	ID
<num>	NUM
<声明>	declare
<声明列表>	declare_list
<内部声明>	inner_declare
<内部变量声明>	inner_var_declare
<函数声明>	func_declare
<参数>	para
<参数项>	param
<参数列表>	para_list
<实参列表>	argu_list
<语句>	stc
<语句串>	stc_list
<语句块>	stc_block
<if 语句>	if_stc
<while 语句>	while_stc
<return 语句>	return_stc
<赋值语句>	assign_stc
<表达式>	exp
<加法表达式>	add_exp
<项>	item
<因子>	factor
<数组>	array

非终结符和助记符的关系

3.2.2 语法分析部分结构

如图可见语法分析部分结构图。其中分析表生成器是关键。分析表生成器首先求出 First 集和 Follow 集，判断其是否符合 SLR(1)文法规则，如符合则继续利用构造的 First 集、Follow 集构造 DFA，生成 SLR(1)分析表。

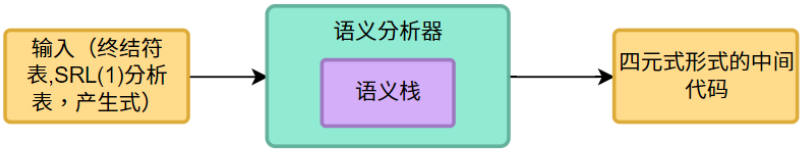


语法分析部分结构图

状态	ACTION				GOTO			
	终结符 1	终结符 2	…	终结符 n	终结符 1	终结符 2	…	终结符 n
0								
1								
…								
n								

SLR(1)分析表

3.2.3 语义分析部分结构



语义分析部分结构图

Sm	Xm.VAL	Xm
Sm-1	Xm-1.VAL	Xm-1
. .	. .	. .
S0	/	#
STATE	VAL	SYM

语义栈

3.2.4 各类语句语法规则和语义流程

- 变量声明语句

语法规则：

declare ::= int ID ;	inner_var_declare ::= int ID
inner_var_declare ::= int ID [factor]	
inner_var_declare ::= int ID [factor] [factor]	

语义流程：

a）使用语义规则获取变量的类型，大小。

b）创建一个新的数据项，保存该变量的名称，类型，当前所在语句块的位置到变量表 varTable 中。

c）如果声明的是数组，创建一个新的数据项，保存数组的行列大小、维数，当前所在语句块的位置到变量表 arrTable 中。

- 赋值语句

语法规则：

assign_stc ::= ID = exp ;	exp ::= add_exp
add_exp ::= item	add_exp ::= item + add_exp
add_exp ::= item - add_exp	item ::= factor
item ::= factor * item	item ::= factor / item
factor ::= NUM	factor ::= (exp)
factor ::= ID	factor ::= array
assign_stc ::= array = exp	

语义流程：

a）检查表达式左端变量是否在符号表中，不存在则进行错误处理。

b）进行中间代码生成

找到赋值语句左端的变量

新建一个中间变量

基于表达式 exp，得到对 n 赋值的中间代码

将计算得到的 n 的值赋给 s

- 条件控制语句

语法规则 1：

if_stc ::= if (exp) A stc_block

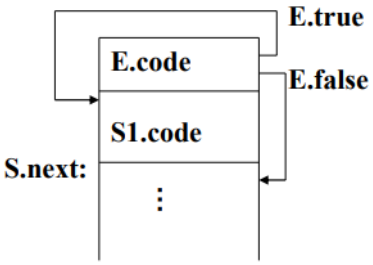
语义流程 1：

a）获得 stc_block 在状态栈中的内容和 exp 在状态栈中的内容

b）获得真值跳转地址和结束地址

c）计算条件表达式，生成相应的中间代码

d）判断是否表达式是否为真，若为真，设置真值跳转地址，并跳转。若为假，跳转到结束地址



语法规则 2：

```
if_stc ::= if ( exp ) A stc_block N else M A stc_block
```

语义流程 2：

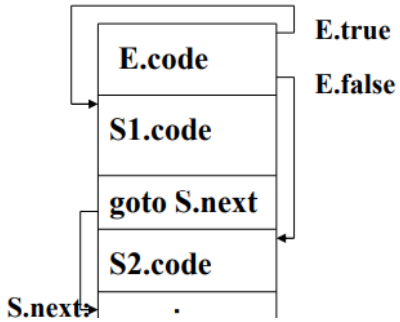
a) 获得 stc_block 在状态栈中的内容和 exp 在状态栈中的内容

b) 获得真值跳转地址和假地址跳转地址、结束地址

c) 计算条件表达式，生成相应的中间代码

d) 判断是否表达式是否为真，若为真，设置真值跳转地址，并跳转。若为假，跳转到假地址

e) 设置跳转到结束地址的指令



- 循环控制语句

语法规则：

```
while_stc ::= while M ( exp ) A stc_block
```

语义流程：

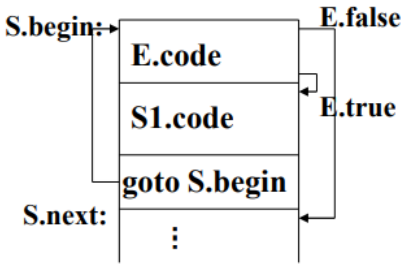
a) 建立分支入口：true、false、begin

b) 获得条件表达式

c) 跳转到 begin 地址，生成相应的中间代码

d) 若条件表达式为真，跳转到 true 地址;为假，跳转到 false 地址

e) 最后，跳转到 begin 地址



- 过程调用

语法规则：

factor ::= ID (argu_list)

语义流程：

a）检查 FuncTable 中 ID 是否存在

b）获得 argu_list

c）生成中间代码

- 数组

语法规则：

array ::= ID [factor]	array ::= ID [factor] [factor]

语义流程：

默认数组按行存放

a）检查 arrTable 中 ID 是否存在

b）计算基地址和位移

c）生成中间代码

3.2.5 代码优化

划分了基本块。

划分基本块算法如下：

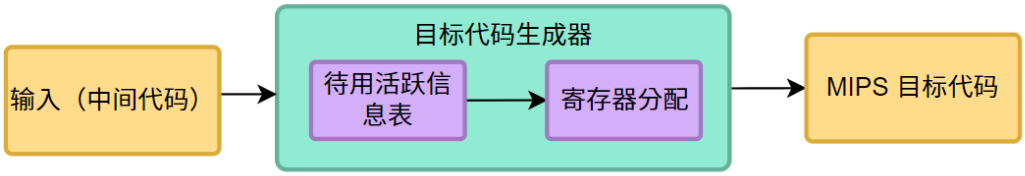
1. 求出四元式程序中各基本块的入口语句，它们是:(1) 程序的第一个语句，或 (2) 由条件或无条件转移语句转到的语句，或 (3) 紧跟在条件转移语句后面的语句;

2. 构造每一个入口语句所属的基本块，它们是从入口语句到: (1) 下一个入口语句(不包括它)，或 (2) 一转移语句(包括它)，或 (3) 一停语句;

3. 未纳入任一基本块的语句为控制无法到达的语句，可予删除。

3.3 目标代码生成器

3.3.1 目标代码生成器结构



目标代码生成器

3.3.2 目标代码生成算法

(1) 找到各基本块的入口活跃和出口活跃变量

① 正向分析每个基本块，分别找到其使用但未定义的变量集合 use 和定义变量集合 def

② 初始化每个基本块的入口活跃变量为 use，出口活跃变量为空

③ 对每一基本块，若它下一基本块的入口活跃变量不存在于该块的出口活跃变量中，则将其插入

(2) 计算待用信息和活跃信息

① 初始化: 对基本块中各变量的待用信息栏置：非待用(^)

活跃信息栏置：非活跃(^) 或 活跃(Y)

② 从基本块的出口至入口处理各个四元式 (i) A:=B op C

把符号表中 A 的待用-活跃信息附加到四元式 i 的左部

把符号表中 A 的待用-活跃信息置为 (^, ^)

把符号表中 B 和 C 的待用 - 活跃信息附加到 i 的左右操作数

把符号表中 B 和 C 的待用 - 活跃信息置为 (i , Y)

③ 重复步骤 2 直到所有中间代码处理完。

序号	四元式	左部	左操作数	右操作数

四元式表

名字	...	待用信息	活跃信息

符号表

(3) 代码生成

寄存器描述数组 RVALUE, RVALUE 中存储寄存器分配给的变量；地址描述数组 AVALUE 中存储变量在某寄存器中或某寄存器和地址中或某地址中

对基本块中每个四元式 (i) A:=B op C

① 为 A 分配寄存器 R, 以(i) A:=B op C 为参数调用函数过程 GETREG(i: A:=B op C) 返回 R,放 A 的当前值;

② 由 AVALUE[A]和 AVALUE[B]确定 B 和 C 的当前值存放位置，设为 B‘和 C’；（优先取寄存器中的值）

③ 生成目标代码: 若 B‘≠R , 则生成 LD R, B’ op R, C‘ 否则, 生成 op R, C’
如果 B‘或 C’是 R，则删除 AVALUE[B]或 AVALUE[C]中 R

④ 令 AVALUE[A] = { R } RVALUE[R] = { A } (5) 若 B 和 C 为 (^, ^) , 则释放其占用的 R

⑤ 处理完基本块中所有四元式后, 对仅在 R 中且出口活跃的 变量 M 生成 ST R, M

其中，寄存器分配算法如下

函数过程: GETREG(i: A:=B op C)

功能: 输入四元式(i) 及其待用 - 活跃信息, 输出存放 A 的 R

算法: ① 若 B 的当前值在 Ri 中,且 RVALUE[Ri]仅含 B, 此外，或 B 与 A 是同一个标识符或 B 在 i 后不活跃, 则选取 Ri 为 R, 转 4.

② 若有空闲寄存器, 则选取一个 Ri 为 R, 转 4.

③ 从已分配的寄存器中选取一个 Ri 为 R, 使其满足:

- 占用 Ri 的变量同时在内存, 或
- 在基本块中最远处被引用或不引用

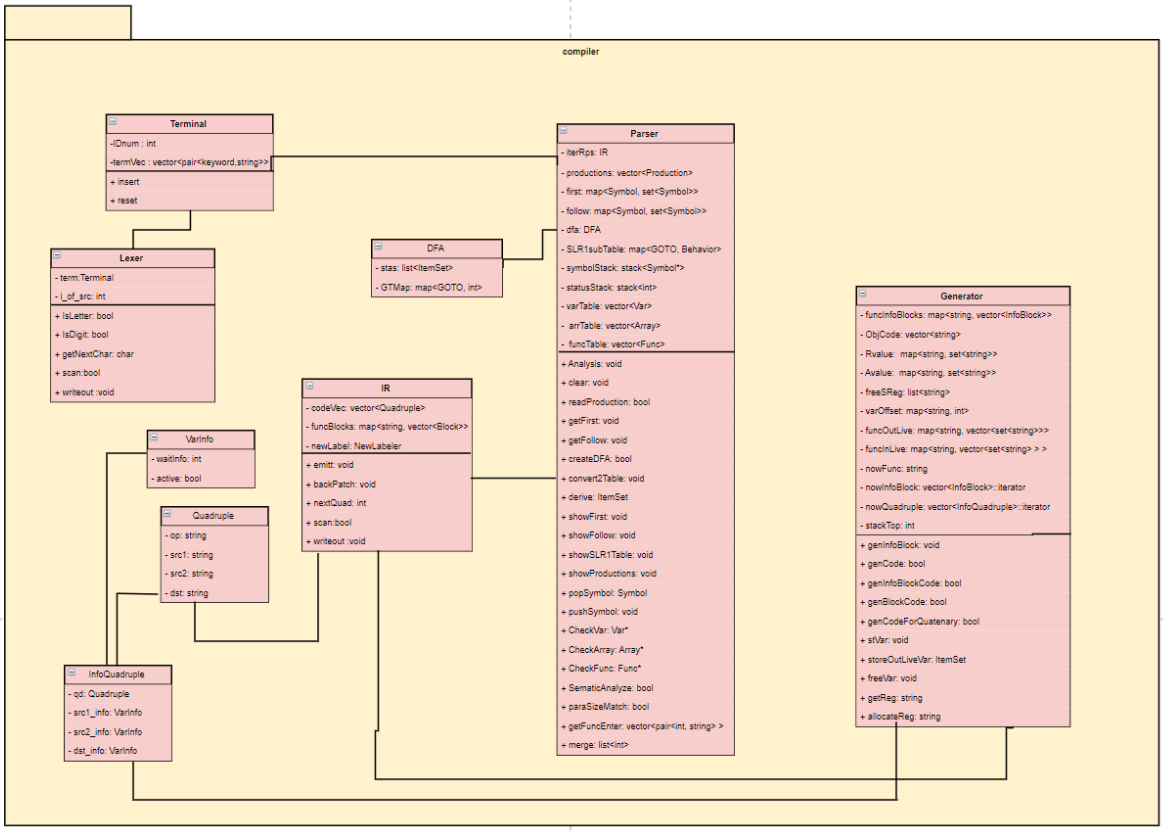
确定 Ri 后的处理: 对 RVALUE[Ri]中的每一个变量 M

- 若 AVALUE[M]不含有 M, 则生成 ST Ri, M
- 若 M 是 B, 或 M 是 C 且 B 在 RVALUE[Ri]中 则令 AVALUE[M]={M, Ri}, 否则令 AVALUE[M]={M}
- 删除 RVALUE[Ri]中的 M

④ 即给出 R, 返回

四、程序具体实现

4.1 类设计图



类设计图

4.2 词法分析器

4.2.1 类设计

```
class Lexer{
public:
    Terminal term;
    Lexer(){
        i_of_src=0;
    }
    void lexer(MainWindow *mw);
private:
    int i_of_src;
    bool IsLetter(char ch);
    bool IsDigit(char ch);
    char getNextChar(string src);
    bool scan(string src,MainWindow *mw);
    void writeout(int type,int num,MainWindow *mw);
                                    };
```

词法分析器 Lexer 类

词法分析器主要是对输入字符串进行处理，与上一次提交的词法分析器不同的是，这次的词法分析器使用了朴素的字符串处理方式，对输入文本进行逐词处理，将识别的字符存入 Terminal 终结符类中，供语法分析器查找。分析过程成功 scan 返回 1，否则返回 0。

```
    class Terminal{
    public:
        int IDnum;
        vector<pair<keyword,string>> termVec;
        void insert(keyword kd,string x,bool isID=false){
            this->termVec.push_back({kd,x});
            if(isID){
                this->IDnum++;
            }
        }
        Terminal(){
            this->IDnum=0;
        }
        void reset(){
            this->IDnum=0;
            this->termVec.clear();
        }
                                        };
```

终结符类 Terminal

4.3 语法和语义分析器

4.3.1 类设计

```
class Parser{
public:
    Parser(Terminal tm,MainWindow *mw);//初始化，语法分析，语义分析
    friend bool operator ==(const set<pair<string,string>>&one, const
set<pair<string,string>>& other);
     IR iterRps;

private:
    //syntax anlysis 语法分析
    vector<Production>productions;
    map<Symbol, set<Symbol>> first;//由 Parse.conf 构造出的 first 集
    map<Symbol, set<Symbol>> follow;
    DFA dfa;
    map<GOTO, Behavior> SLR1subTable;
    map<int,set<pair<string,string>>> SLR1_table;
    bool readProduction(Terminal tm,MainWindow *mw);
```

```
    void getFirst();
    void getFollow();
    bool createDFA(MainWindow *mw);
    void convert2Table();
    ItemSet derive(Item item);
    void showFirst(MainWindow *mw);
    void showFollow(Terminal tm,MainWindow *mw);
    void showSLR1Table(MainWindow *mw);
    void showProductions();
    //symatic anlysis 语义分析
    stack<Symbol*> symbolStack;
    stack<int> statusStack;
    vector<Var> varTable;
    vector<Func> funcTable;
    NewTemper newTemper;
    Symbol* popSymbol();
    void pushSymbol(Symbol* sym,MainWindow *mw);
    Func* CheckFunc(string ID);
    Var* CheckVar(string ID);
    bool SematicAnalyze(Terminal tm,MainWindow *mw);
    bool paraSizeMatch(list<string>&argument_list,
list<DataType>&parameter_list);
    vector<pair<int, string> > getFuncEnter();
    list<int> merge(list<int>&l1, list<int>&l2);
};
```

语法和语义分析器类 Parser

```
class IR //intermediate representation
{
private:
    vector<Quadruple> codeVec; //存放四元式组
    map<string, vector<Block> >funcBlocks; //存放函数块
    NewLabeler newLabel;

public:
    void emitt(Quadruple q);// 产生一个四元式并将其存入 codeVec
    void backPatch(list<int>nextList,int quad);// 将 quad 回填至
nextList 对应的四元式中
        int nextQuad();//获得新的地址
    void processBlocks(vector<pair<int, string>> funcEnter);
    void output(MainWindow* mw);
};
```

中间代码类 IR

```
class Symbol{
public:
    bool isTerm;
    string content;
    friend bool operator ==(const Symbol& one, const Symbol& other);
    friend bool operator < (const Symbol& one, const Symbol& other);
    Symbol(const Symbol& sb);
    Symbol(const bool& isTerm, const string& content);
    Symbol(){}
      };
class Id :public Symbol
{
public:
    string name;
    Id(const Symbol& sym, const string& name);
};

class Num :public Symbol
{
public:
    string number;
    Num(const Symbol& sym, const string& number);
};

class FuncDeclare :public Symbol
{
public:
    list<DataType>funcList;
    FuncDeclare(const Symbol& sym):Symbol(sym) {};
};

class Parameter :public Symbol
{
public:
    list<DataType>plist;
    Parameter(const Symbol& sym):Symbol(sym) {};
};

class ParaList :public Symbol
{
public:
    list<DataType>plist;
    ParaList(const Symbol& sym):Symbol(sym) {};
};
```

```
class StcBlock :public Symbol
{
public:
    list<int>nextList;
    StcBlock(const Symbol& sym):Symbol(sym) {};
};

class StcList :public Symbol
{
public:
    list<int>nextList;
    StcList(const Symbol& sym):Symbol(sym) {};
};

class Sentence :public Symbol
{
public:
    list<int>nextList;
    Sentence(const Symbol& sym):Symbol(sym) {};
};

class WhileStc :public Symbol
{
public:
    list<int>nextList;
    WhileStc(const Symbol& sym):Symbol(sym) {};
};

class BoolStc :public Symbol
{
public:
    list<int>falseList;
    BoolStc(const Symbol& sym):Symbol(sym) {};
};

class IfStc :public Symbol
{
public:
    list<int>nextList;
    IfStc(const Symbol& sym):Symbol(sym) {};
};

class BoolAnd :public Symbol
{
```

```
public:
    list<int>falseList;
    BoolAnd(const Symbol& sym):Symbol(sym) {};
};

class BoolOr :public Symbol
{
public:
    list<int>falseList;
    BoolOr(const Symbol& sym):Symbol(sym) {};
};

class Exp :public Symbol
{
public:
    string name;
    list<int>falseList;
    Exp(const Symbol& sym):Symbol(sym) {};
};

class M :public Symbol
{
public:
    int quad;
    M(const Symbol& sym):Symbol(sym) {};
    M(){};
};

class N :public Symbol
{
public:
    list<int> nextList;
    N(const Symbol& sym):Symbol(sym) {};
    N(){};
};

class AddExp :public Symbol
{
public:
    string name;
    AddExp(const Symbol& sym):Symbol(sym) {};
};

class Factor :public Symbol
```

```
{
public:
    string name;
    Factor(const Symbol& sym):Symbol(sym) {};
};

class ArguList :public Symbol
{
public:
    list<string> alist;
    ArguList(const Symbol& sym):Symbol(sym) {};
};

class Array :public Symbol
{
public:
    string name;
    string result;
    int dimension;
    string dm1_size,dm2_size;
    Array(string name, int dimension,string dm1_size,string dm2_size);
    Array(const Symbol& sym, const string& name);
    Array(const Symbol& sym):Symbol(sym) {};
};
```

标识符类 Symbol 和他的继承类

```
class Quadruple{
public:
    string op;
    string src1;
    string src2;
    string dst;
    Quadruple(string op,string src1,string src2,string dst)
    Quadruple(){};
};
```

四元式类 Quadruple

```
class Block
{
public:
    string name;//基本块的名称
    vector<Quadruple> codes;//基本块中的四元式
    int next1;//基本块的下一连接块
    int next2;//基本块的下一连接块
    };
```

基本块类 Block

4.3.2 功能函数

4.3.2.1 语法分析部分

- **产生式处理**

涉及函数: readRule

算法思想:

语法分析器先对 production.conf 文件做字符串处理，将其存入产生式 Producion 类中，以” := ”符号作为产生式左右的分割，针对每一个产生式将左部符号存入 left，右部存入 right。

```
class Production{
public:
    int id;
    Symbol left;
    vector<Symbol>right;
};
```

产生式 Production

- **获得 First 集**

涉及函数:

getFirst

算法思想：

设置一个标识符 ChangeFlag，初始时设为 false，该标识符代表非终结符的 First 集合不再改变。针对每一个非终结符，进行下述操作：

1.若 X->a…，则将终结符 a 放入 First（X）中

2.若 X-> ε，则将 ε 放入 First（X）中

3.若有 X->Y1Y2Y3…Yk，则

（1）把 First（Y1）去掉 ε 后加入 First（X）

（2）如果 First（Y1）包含 ε，则把 First（Y2）也加入到 First（X）中，以此类推，直到某一个非终结符 Yn 的 First 集中不包含 ε

（3）如果 First（Yk）中包含 ε，即 Y1~Yn 都有 ε 产生式，就把 ε 加入到 First（X）中

若进行以上三种操作之一，则将 ChangeFlag 置为 true。每次遍历前将其设置为 false，直至遍历所有非终结符后该标志仍为 false 后结束

- **获得 Follow 集**

涉及函数:

getFollow

算法思想：

设置一个标识符 ChangeFlag，初始时设为 false，该标识符代表非终结符的 Follow 集合不再改变。针对每一个非终结符，进行下述操作：

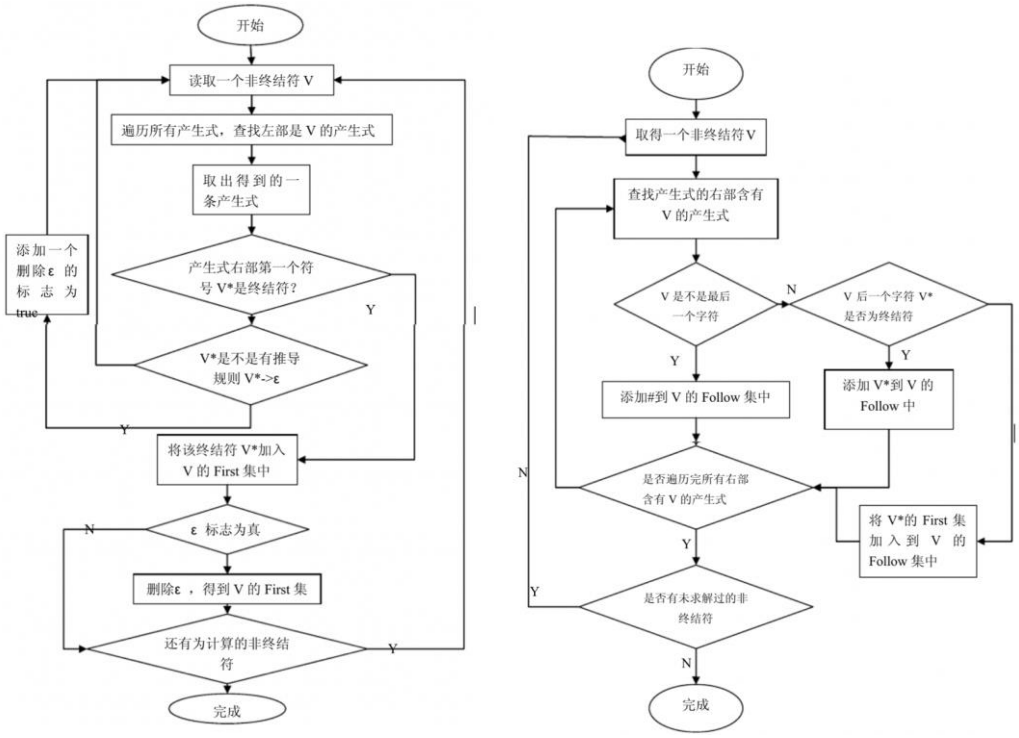
1.对于文法的开始符号 S，置 # 于 Follow (S)中.

2.若 A →aBβ 是一个产生式， 则把 First(β)\|ϵ| 加至 Follow (B) 中.

3.若 A → aB 是一个产生式，或 A →aBβ 是一个产生式而 β⇒ϵ(即 ϵ ∈First

(B), 则把 Follow (A) 加至 Follow (B) 中.

若进行以上三种操作之一，则将 ChangeFlag 置为 true。每次遍历前将其设置为 false，直至遍历所有非终结符后该标志仍为 false 后结束



求 First 和 Follow 集流程图

- **SLR(1)分析表构造**

涉及函数:

createDFA、derive

算法思想:

derive 函数求派生集。

createDFA：

(1)若 $A\rightarrow\alpha\cdot a\beta\in I_k$ ，且 $GO(I_k,a)=I_j$ ，$a\in V_T$ ，则置 $ACTION[k,a]=s_j$;

(2)若 $A\rightarrow\alpha\cdot\in I_k$ ，则对任何终结符 a(包括#)，且满足 a∈Follow (A)时，置 $ACTION[k,a]=r_j$ (j 为产生式 A→α 的编号);

(3)若项目 S'→S·属于 I_k ，则置 ACTION[k,#]=acc;

(4)若 $GO(I_k,A)=I_j$ ($A\in V_N$)，则置 GOTO[k,A]=j;

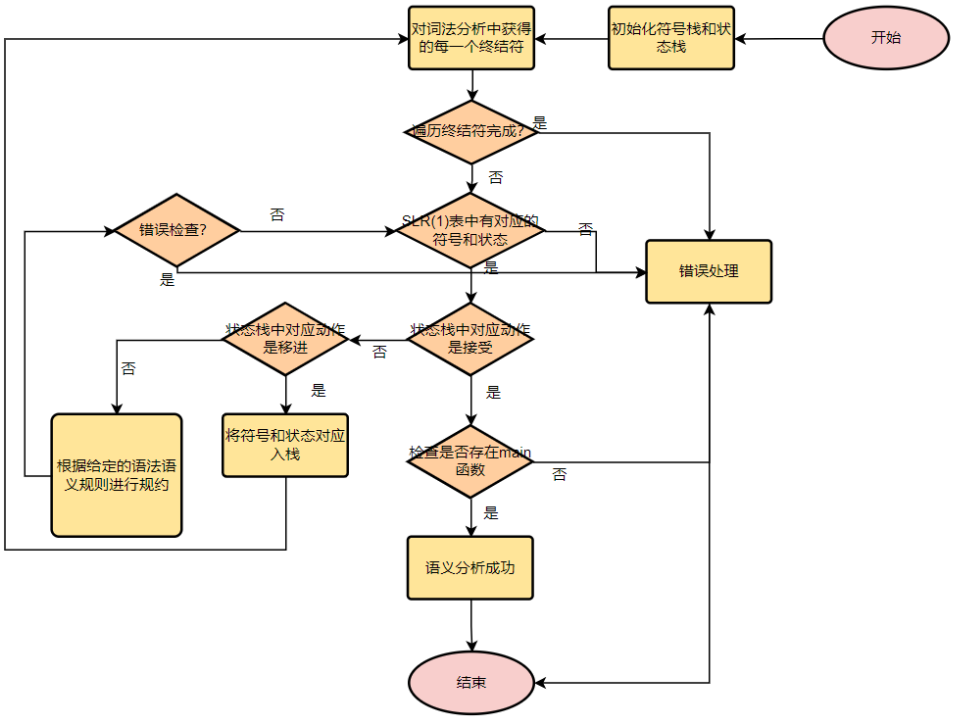
(5)不能用上述方法填入内容的单元均置为“出错标志” (用空白表示)。

```
struct DFA
{
    list<ItemSet> stas;
    map<GOTO, int> GTMap;
};
```

DFA 结构体，其中 ItemSet 为含.的式子的派生集，GOTO 为转移序列

4.3.2.2 语义分析部分

其中 SematicAnalyze 调用其他类成员及函数进行语义分析



语义分析流程图

具体算法详见 3.2.3

4.3.2.3 代码优化/基本块划分部分

涉及函数:

divideBlocks

算法思想:

详见 3.2.5

4.4 目标代码生成器

4.4.1 类设计

```
class Generator{
private:
    map<string, vector<InfoBlock>> funcInfoBlocks;
    vector<string> ObjCode;

    map<string, set<string>>Rvalue;//寄存器描述
    map<string, set<string>>Avalue;//地址描述
```

```
    list<string>freeSReg;// 空闲的寄存器编号$s
    map<string, int>varOffset;//各变量的存储位置
    map<string, vector<set<string> > >funcOutLive;//各函数块中基本块的出口活跃变量集
    map<string, vector<set<string> > >funcInLive;//各函数块中基本块的入口活跃变量集
    string nowFunc;//当前分析的函数
    vector<InfoBlock>::iterator nowInfoBlock;//当前分析的基本块
    vector<InfoQuadruple>::iterator nowQuadruple;//当前分析的四元式
    int stackTop;//当前栈顶

    void genInfoBlock(map<string, vector<Block> >*funcBlocks);
    bool genCode();
    bool genInfoBlockCode(map<string, vector<InfoBlock> >::iterator);
    bool genBlockCode(int nowBaseBlockIndex);
    bool genCodeForQuatenary(int &arg_num, list<pair<string, bool> > &par_list) ;

    void stVar(string reg, string var);
    void storeOutLiveVar(set<string>&outl);
    void freeVar(string var);
    string getReg();
    string allocateReg();
    string allocateReg(string var);

public:
    Generator(IR *pir,MainWindow *mw);
    void outputObjCode(MainWindow *mw);
};
```

代码生成器类 Generator

```
class InfoQuadruple//四元式表上的待用活跃信息
{
public:
    Quadruple qd;
    VarInfo src1_info;
    VarInfo src2_info;
    VarInfo dst_info;
    InfoQuadruple(Quadruple qd, VarInfo src1_info, VarInfo src2_info, VarInfo dst_info);
};
```

带待用活跃信息的四元式类 InfoQuadruple

4.4.2 功能函数

- **找到各基本块的入口活跃和出口活跃变量**

  涉及函数:

  genInfoBlock

  算法思想：

  详见 3.3.2

- **计算待用信息和活跃信息**

  涉及函数:

  genInfoBlock

  算法思想：

  详见 3.3.2

- **代码生成**

  涉及函数:

  genCode //调用 genBlockCode

  genBlockCode//调用 genCodeForQuatenary

  genCodeForQuatenary//为四元式生成目标代码

  stVar //存储变量

  freeVar //释放变量

  getReg //获取寄存器

  allocateReg //分配寄存器

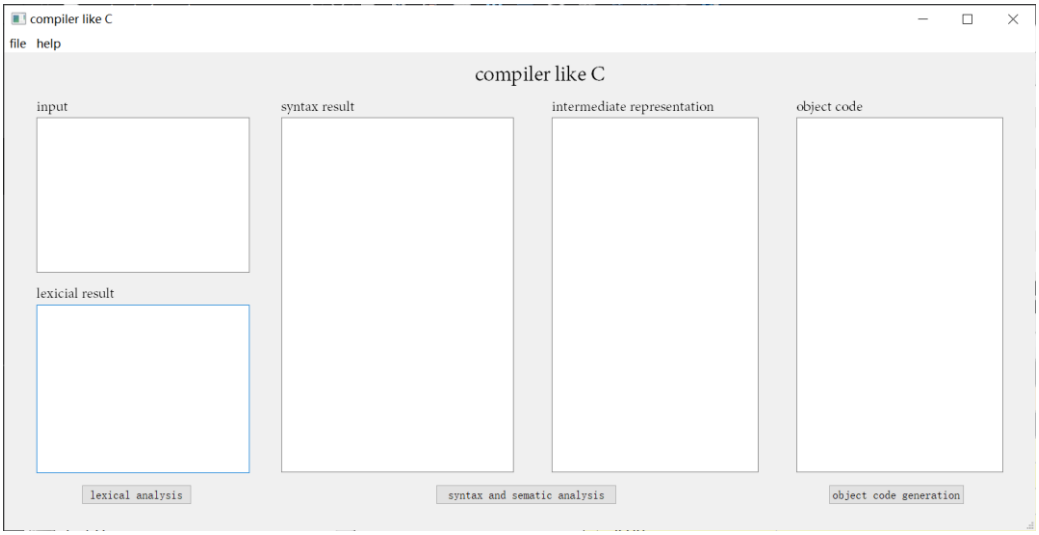
  算法思想：

  详见 3.3.2

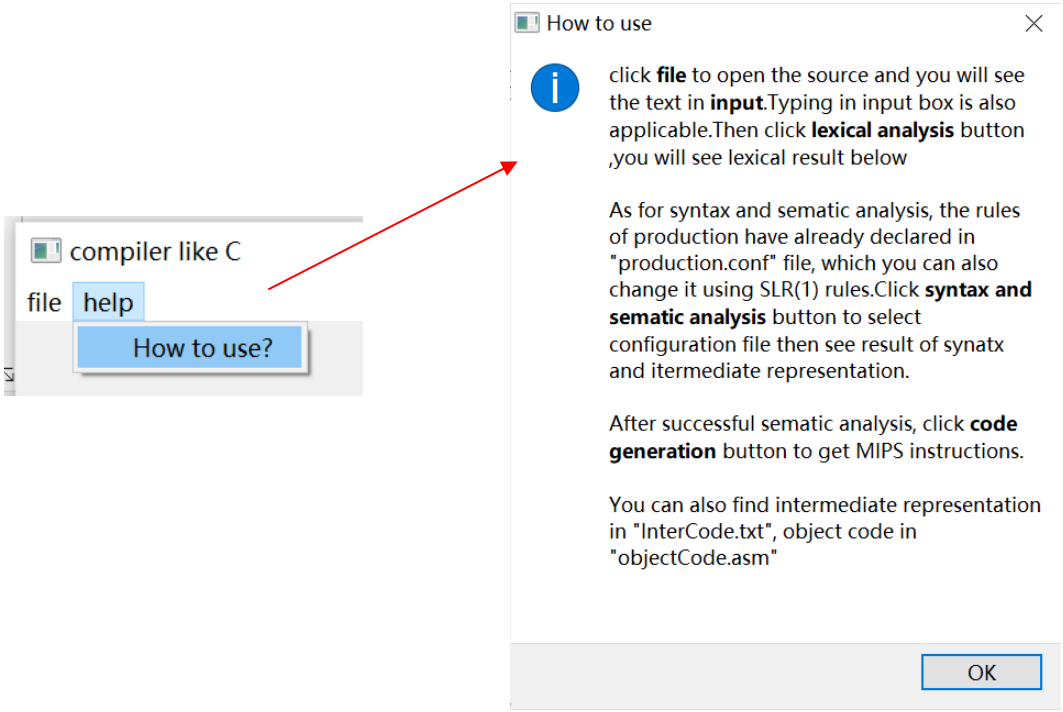
五、执行界面和运行结果截图

5.1 正常运行

进入程序后，界面如下

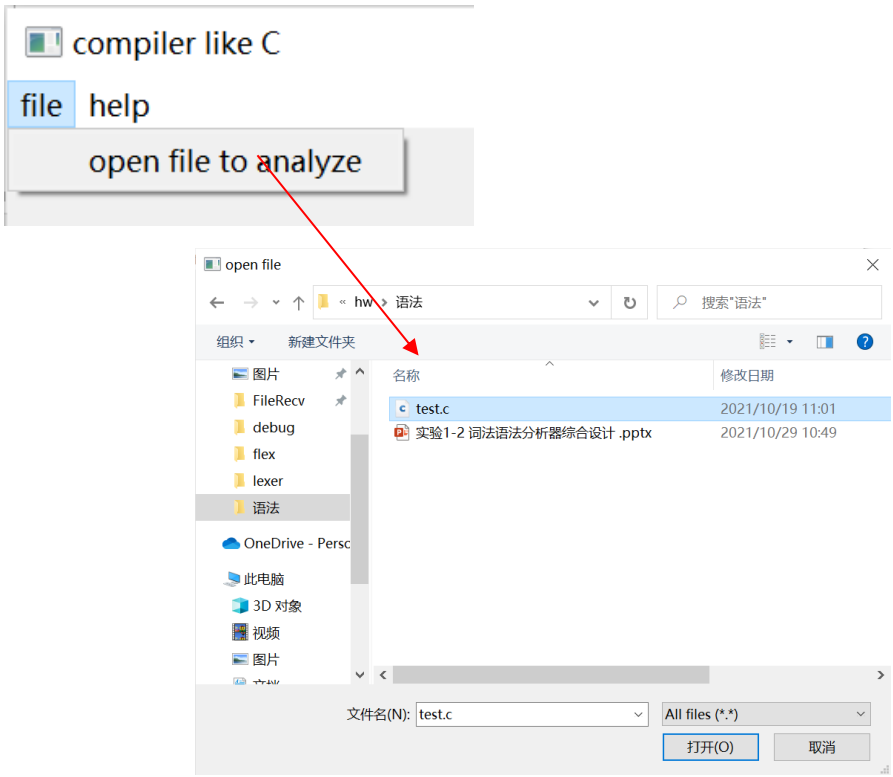


点击菜单栏中 help，有详细的说明介绍



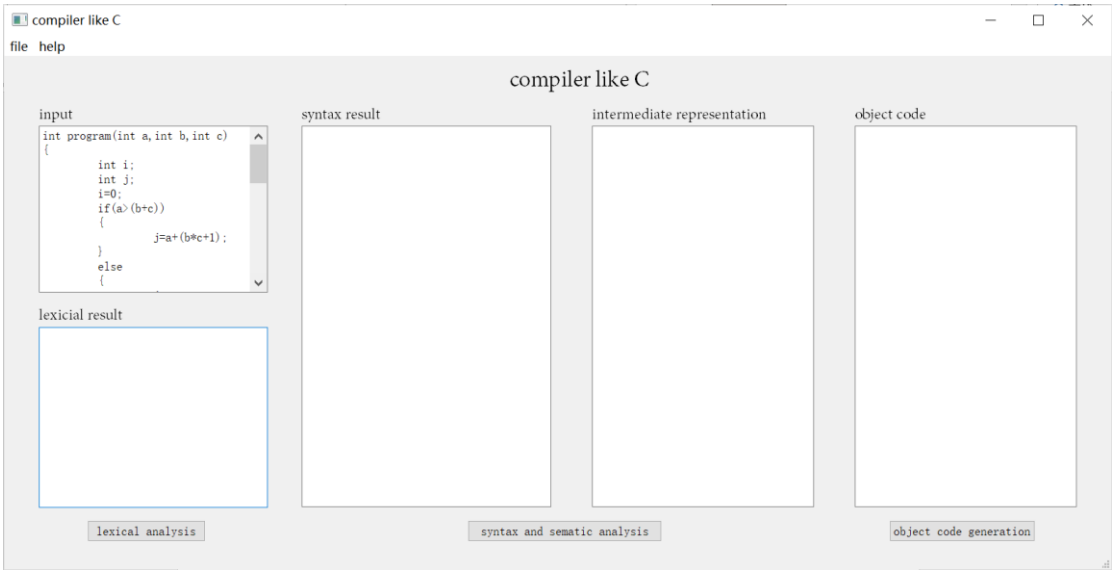
使用说明介绍

用户可以选择直接在 input 框中输入要分析的文本，也可点击菜单栏中的 file，可打开要分析的源文件，这里我们打开 test.c 文件，对其进行词法分析。



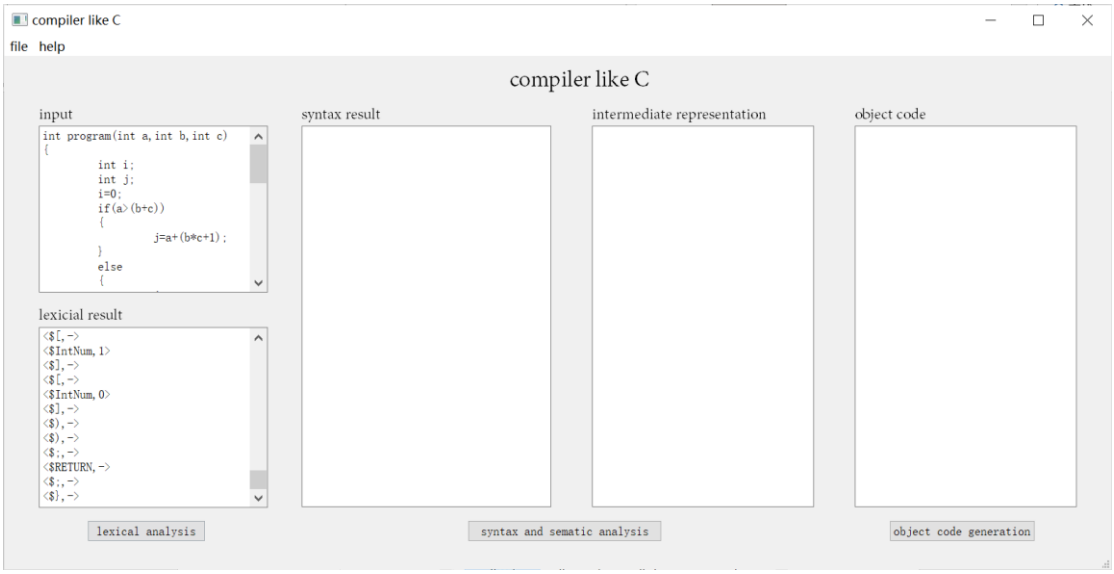
打开文件

此时源文件的内容会显示在 input 框中



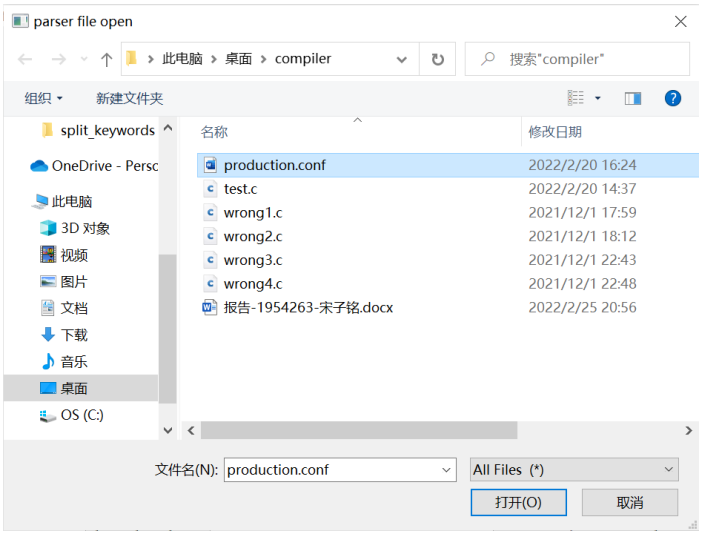
文件的文本内容显示在 input 框中

单击 lexical analysis 按钮即可对源文件进行语法分析，结果显示在 lexical result 中



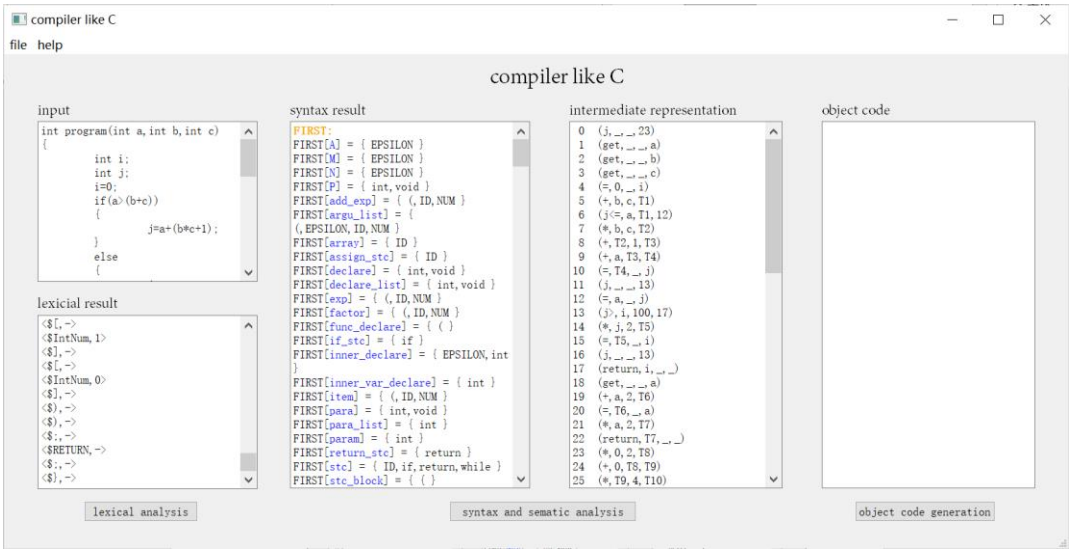
词法分析结果显示在 lexical result 中

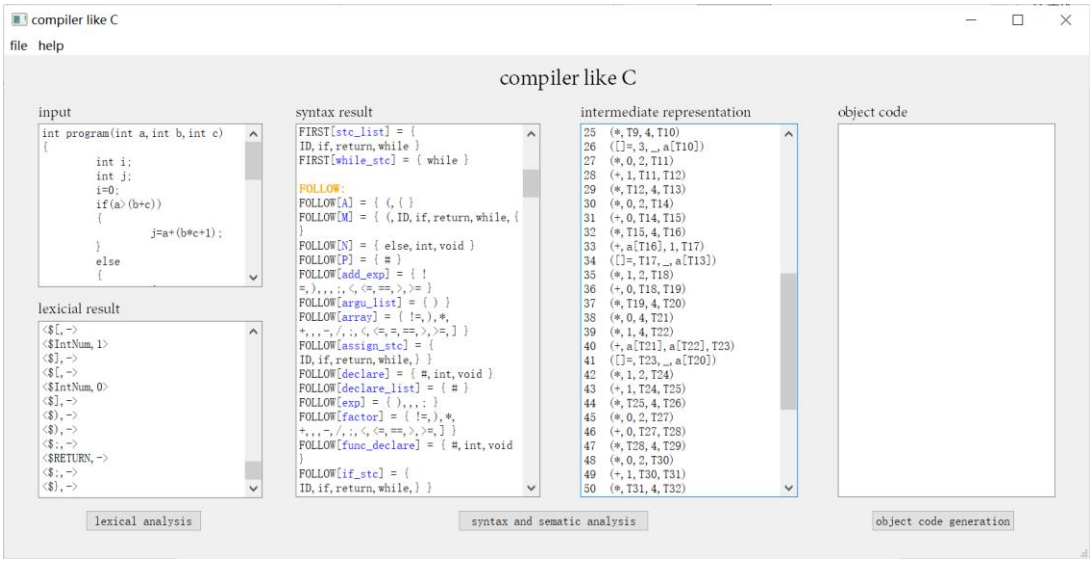
单击 syntax analysis 按钮进行语法分析，可以打开选择的语法文件。注意所选路径中不要含有中文。

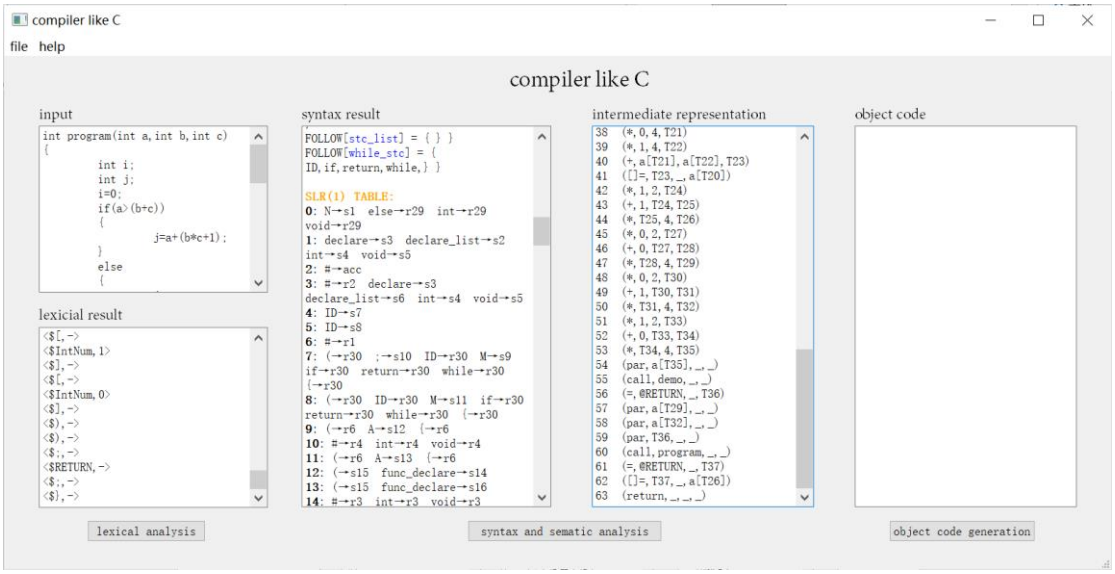


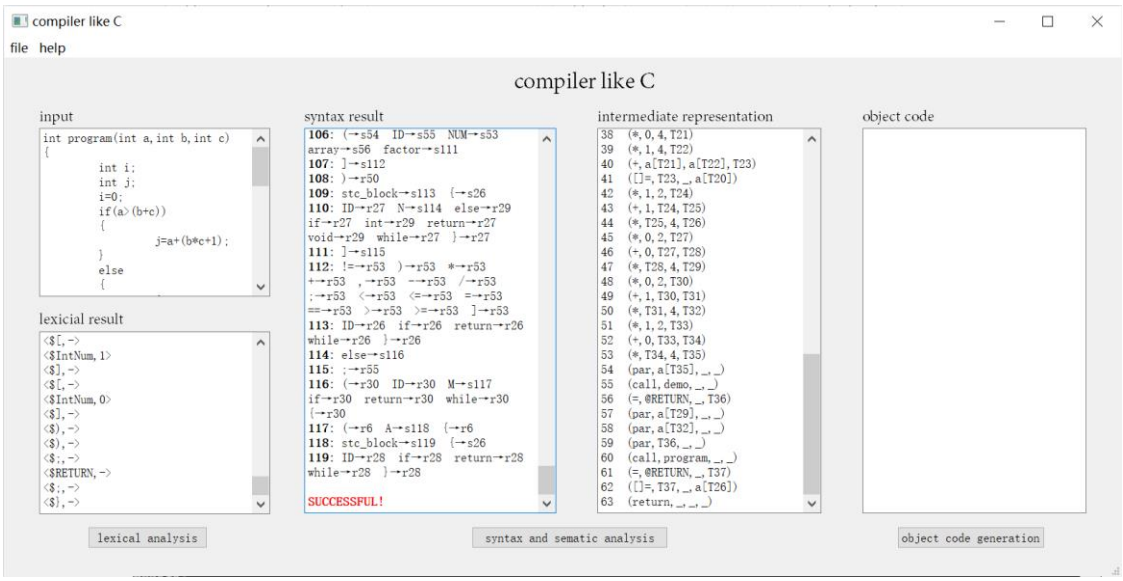
选择语法分析的表达式文件

可识别的语法提前放在了 production.conf 文件中，我们可以看到在该语法能被正确识别情况下的非终结符的 First 集和 Follow 集，SLR(1)分析表。可以看到生成的中间代码。

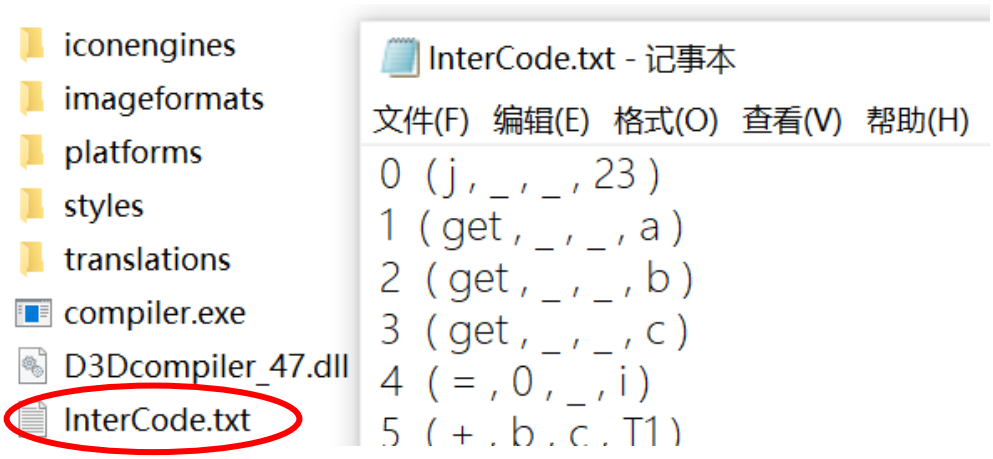






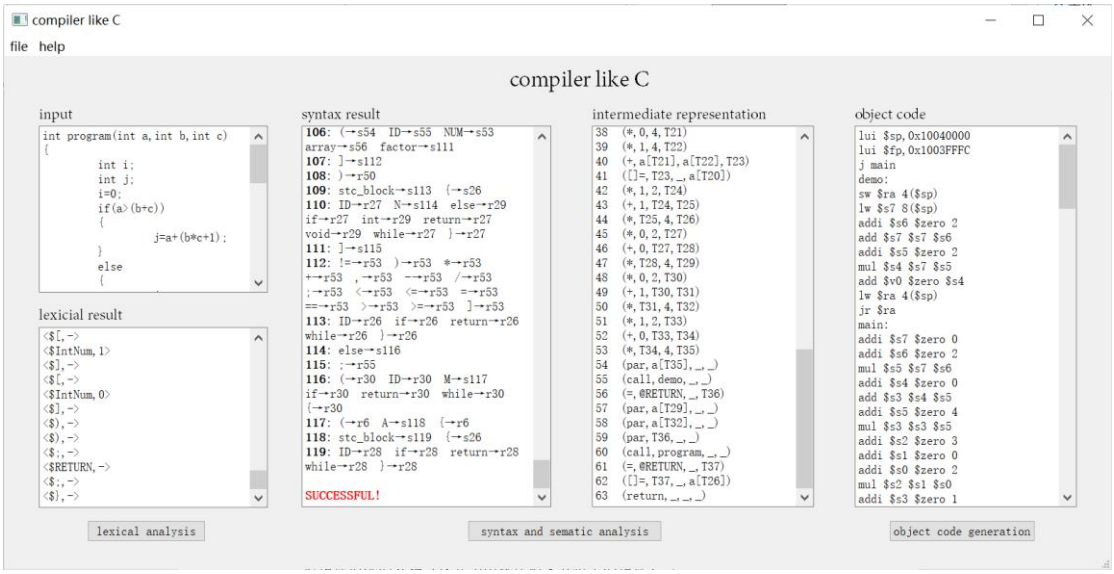


语法分析和语义分析中间代码结果 Qt 展示

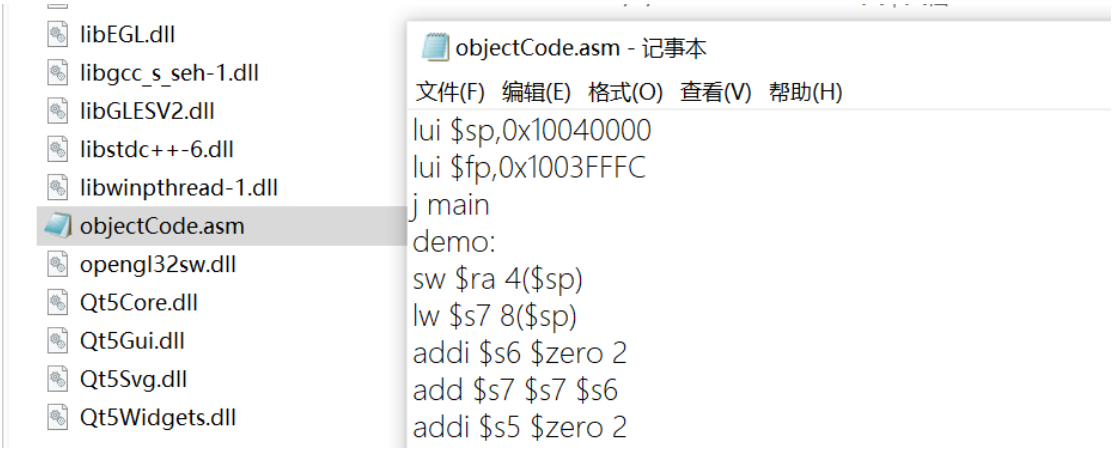


生成的中间代码文件

然后，点击 object code generation 按钮进行目标代码生成。



目标代码生成结果 Qt 展示

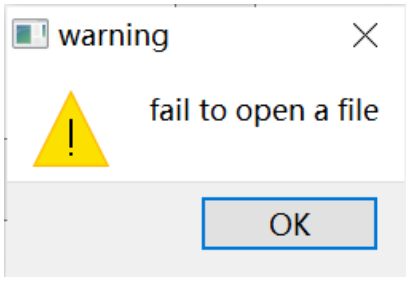


生成的目标代码文件

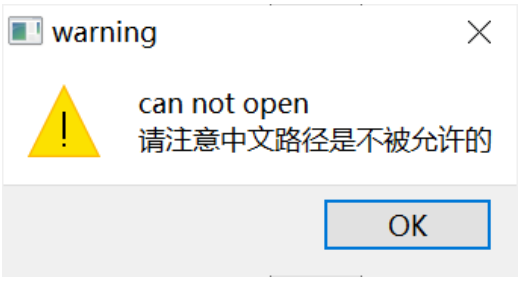
5.2 对错误情况的处理

a. 打开文件失败

出现错误弹窗

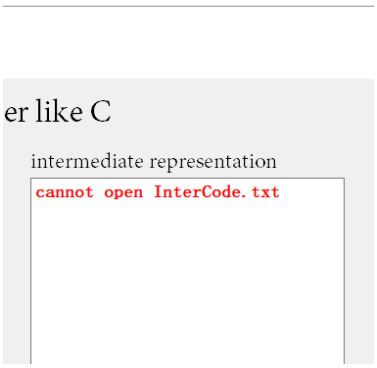


打开源代码失败

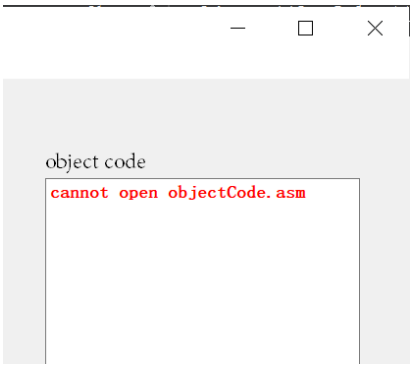


打开产生式文件失败

b. 打开要输入中间代码/目标代码的文件失败



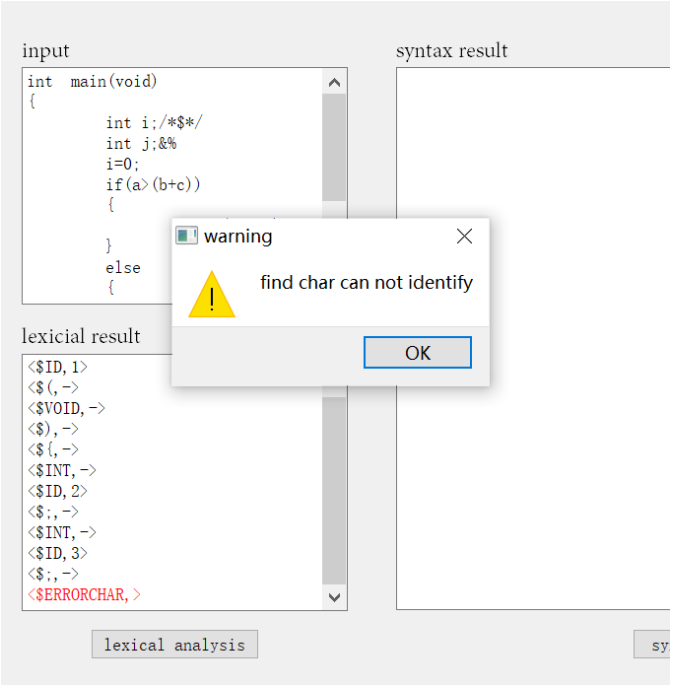
打开要输入中间代码失败



打开要输入目标代码失败

c.词法分析中遇到非法字符

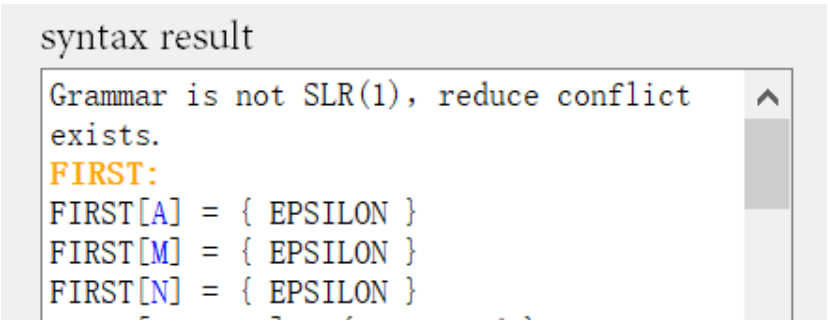
lexical result 中会显示错误字符，弹窗提示词法分析遇到非法字符,输入见 wrong1.c



输入源文件中有非法输入&

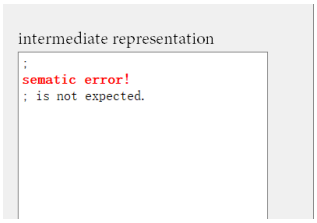
d. 语法分析失败

syntax result 中输出错误信息，但仍可见 First、Follow 集信息



e. 语义分析中遇到不符合语法规则的情况

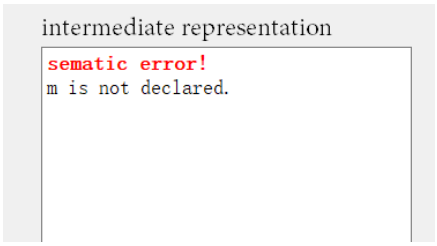
intermediate 框中输出错误提示，具体文件见 wrong2.c



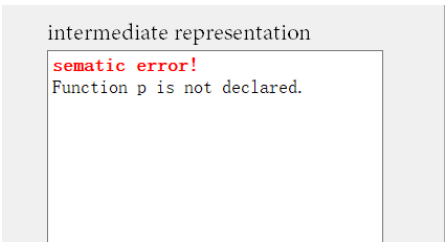
语义分析中遇到不符合语法规则的情况

f. 使用未声明变量/函数

语义分析中遇到未定义的函数，intermediate 框中输出错误提示，具体文件见 wrong3.c,wrong4.c



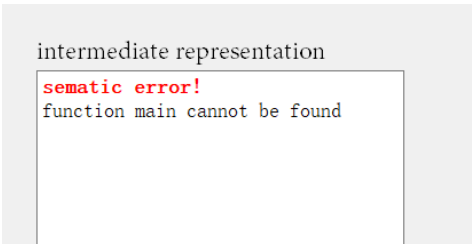
使用未声明变量



使用未声明函数

g. 缺少 main 函数

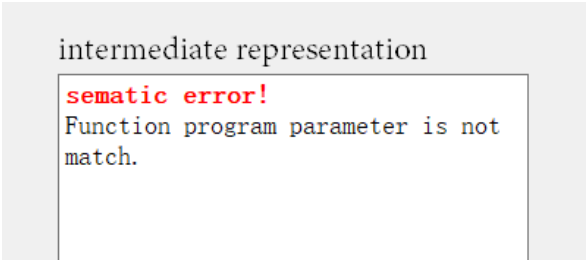
在输出中间代码的文本框中输出提示，具体文件见 wrong5.c



缺少 main 函数

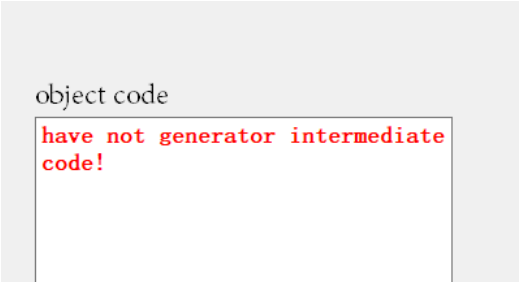
h. 形参与实参个数不匹配

在输出中间代码的文本框中输出对应提示



形参与实参个数不匹配

i. 未进行中间代码生成而直接进行目标代码生成



六、设计中遇到的问题及解决方法

- 语法规则的改进

项目要求中只给出了要识别的语法，并未给出具体的产生式。我的语法分析器是SLR(1)的，要将给出的语法先转换成对应的文法。尤其是在增加数组部分以后，由于语义分析时还要依照输入的产生式进行规约，这关系到四元式的生成，所以其产生式的构造就更为重要。

数组在声明时要将其的行列大小、起始地址等保存起来。如果写成如下产生式，规约 *a[1]=0;*和 *int a[5]* 时对数组都使用(1)产生式进行规约，无法区分声明变量和引用变量。

```
array ::= ID [ factor ]                    (1)
inner_var_declare ::= array                (2)
assign_stc ::= array = exp ;               (3)
```

而将产生式修改为如下形式

```
array ::= ID [ factor ]                    (1)
inner_var_declare ::= int ID [ factor ]    (2)
assign_stc ::= array = exp ;               (3)
```

规约 *a[1]=0;*使用(1)产生式;规约 *int a[5]* 时使用(2)产生式，区分出声明变量和引用变量，在语义分析时可区分二者，分别进行分析。

- 入口活跃变量和出口活跃变量的求解

课堂上并没有讲解如何处理这一问题，我参考了一些网上的资料并结合个人理解，入口活跃变量就是那些在本基本块未被定义却被使用的变量，出口活跃变量就是在下一基本块中活跃的入口变量且在该基本块中定义或使用过。

- 越界问题

整个项目中，出现了许多的边界条件，为空的情况，比如输入的原文件为空、产生的中间代码为空等等，这就需要对这些数组/容器的边界条件，指针的为空情况进行处理，否则就会导致程序的崩溃。

```
Desktop_Qt_5_12_10_MinGW_64_bit-Release\release\compile.exe ...
21:21:32: 程序异常结束。
21:21:32: The process was ended forcefully.
21:21:32: C:\Users\1234\Documents\build-compile-Desktop_Qt_5_12_10_MinGW_64_bit-
Release\release\compile.exe crashed.
```

程序崩溃

对越界问题的处理并不复杂，只需增加对边界条件的判断，难的在于找出这些越界点，需要增加测试程序。

- 打包问题

打包时文件路径即为打包时文件所在的路径。而使用者在使用时往往和打包路径不一致，于是我所有的文件都采用了用户手动选择的方法来进行输入，而不是使

用相对路径查找。

七、设计体会

这次课程设计帮助我更好的理解掌握了课上学习的算法知识，增强了我的实践能力，按照 PPT 上的内容讲解和算法可以逐步的实现编译器的设计。但在细节和类设计上需要多下写功夫。

设计过程中最复杂的部分就在于类设计和协调多个类之间的关系。开始时我将语法和语义分析器分开设计但是发现二者使用的许多变量都是相同的，联系也很紧密，于是将其合为了一类。中间代码在语法和语义分析器中生成，但在目标代码生成器中又要被再次使用，采取了友元类的方法。

在设计中，也进一步学习了 MIPS 指令的相关知识，详细了解了各个寄存器的作用，感觉和硬件所学知识结合了起来，整体感觉收获很大。