

实验 2 中间代码生成器设计说明

一、运行和开发环境

- ①无图形界面：Win10 下 Visual Studio 开发，通过.exe 可执行文件在 cmd 窗口下运行。
- ②图形化界面：Win10 下 Qt5 开发，通过.exe 可执行文件由窗体程序运行。

二、功能

由于本中间代码生成器的设计基于实验 1 中的词法语法分析器设计而成，所以从词法分析器和语法分析器开始分别说明。

1、词法分析器

能识别的单词：

- ✓ 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：#

扩充单词：

除上述作业要求单词以外，还扩充了如下单词：

- ✓ 关键字：char | const | unsigned | bool | true | false
- ✓ 左方括号：[
- ✓ 右方括号：]
- ✓ 单引号：'
- ✓ 双引号："

2、语法分析器

能分析的文法：

本语法分析器主要采用 LL(1)文法：

- ✓ Program ::= <类型> <ID> '(' ' ' <语句块>
- ✓ <类型> ::= int | void
- ✓ <ID> ::= 字母(字母|数字)*
- ✓ <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> (注：{ } 中的项表示可重复若干次)

- ✓ $\langle \text{语句串} \rangle ::= \langle \text{语句} \rangle \{ \langle \text{语句} \rangle \}$
- ✓ $\langle \text{语句} \rangle ::= \langle \text{if 语句} \rangle | \langle \text{while 语句} \rangle | \langle \text{return 语句} \rangle | \langle \text{赋值语句} \rangle$
- ✓ $\langle \text{赋值语句} \rangle ::= \langle \text{ID} \rangle = \langle \text{表达式} \rangle ;$
- ✓ $\langle \text{return 语句} \rangle ::= \text{return} [\langle \text{表达式} \rangle]$ (注: [] 中的项表示可选)
- ✓ $\langle \text{while 语句} \rangle ::= \text{while} (' \langle \text{表达式} \rangle ') \langle \text{语句块} \rangle$
- ✓ $\langle \text{if 语句} \rangle ::= \text{if} (' \langle \text{表达式} \rangle ') \langle \text{语句块} \rangle [\text{else} \langle \text{语句块} \rangle]$ (注: [] 中的项表示可选)
- ✓ $\langle \text{表达式} \rangle ::= \langle \text{加法表达式} \rangle \{ \text{relop} \langle \text{加法表达式} \rangle \}$ (注: relop $\rightarrow \langle | < = | > = | ! = \rangle$)
- ✓ $\langle \text{加法表达式} \rangle ::= \langle \text{项} \rangle \{ + \langle \text{项} \rangle | - \langle \text{项} \rangle \}$
- ✓ $\langle \text{项} \rangle ::= \langle \text{因子} \rangle \{ * \langle \text{因子} \rangle | / \langle \text{因子} \rangle \}$
- ✓ $\langle \text{因子} \rangle ::= \text{ID} | \text{num} | (' \langle \text{表达式} \rangle ')$

补充的功能:

对语法进行了错误处理, 归约到错误步骤后, 报出错误, 并指出发生错误的归约步骤。

3、中间代码生成器

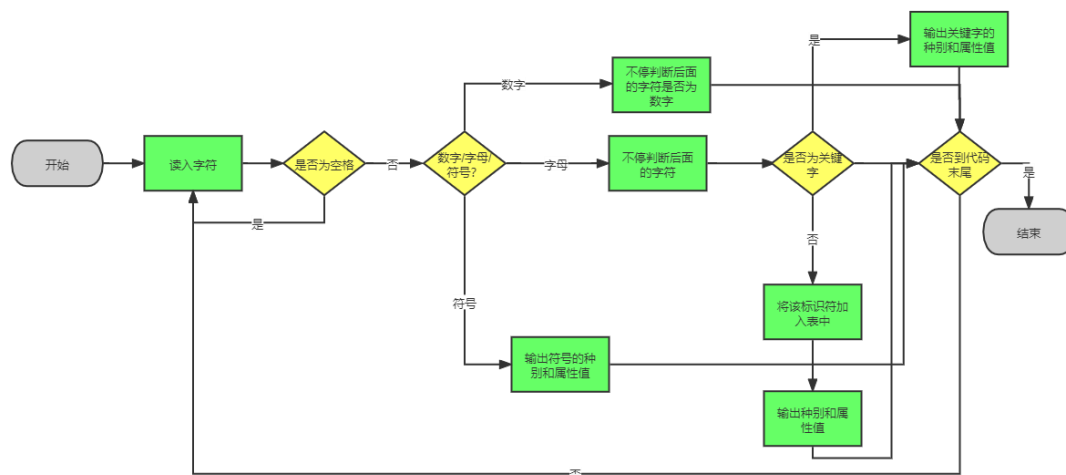
在前面实验的基础上 (词法、语法分析), 进行语义分析和中间代码生成器的设计, 输入源程序, 输出等价的中间代码序列, 并以四元式的形式作为中间代码。

添加了静态语义错误的诊断和处理。

三、主程序框图

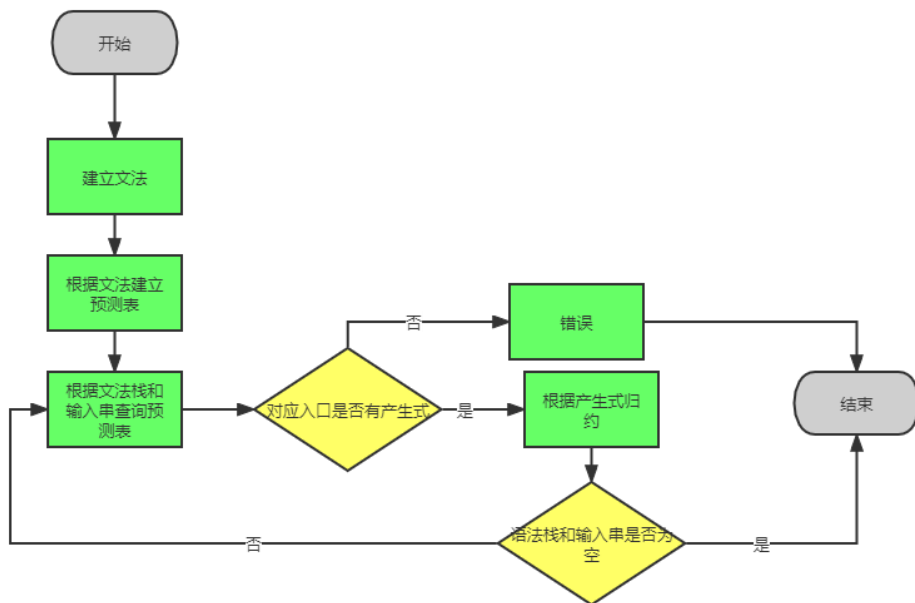
词法分析器程序框图:

主要思路: 用 scanner 函数依次扫描输入的源代码, 以一个或多个空格为间隔, 遇到不同的标识符和运算符进行不同的处理, 对其进行分类, 并输出相应的结果。



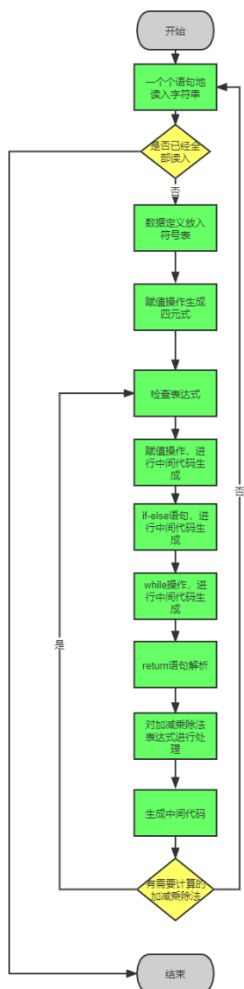
语法分析器程序框图:

主要思路: 根据文法分析器对源代码的文法分析, 将文法分析结果转化为语法分析的终结符, 并使用 LL(1)文法, 将文法转化为左递归的文法, 创建语法栈和字符串栈, 并步步归约, 直到发生错误或者两个栈都已空, 结束语法分析: 如果错误, 报错; 如果没有错误, 则规约成功。



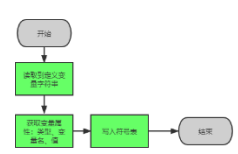
中间代码生成器的程序框图：

主要思路：在词法语法分析器的结果通过的情况下，再对源代码的字符串进行分析。对满足指定结构的字符串进行处理，并根据字符串的处理结果进行中间代码四元式的生成。

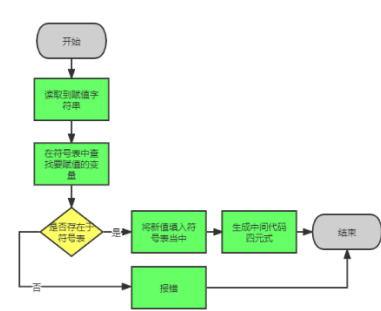


具体模块的程序框图：

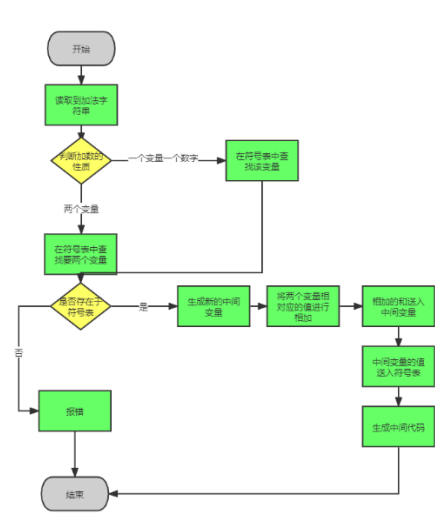
定义模块：



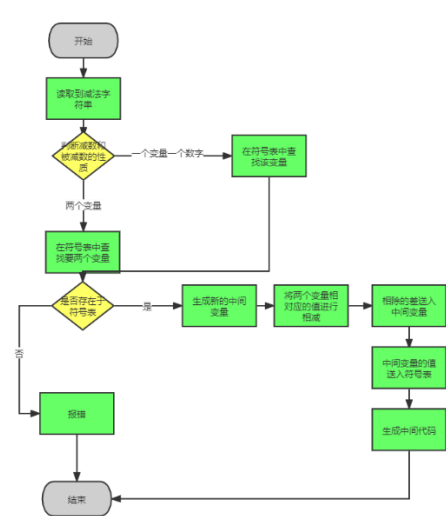
赋值模块：



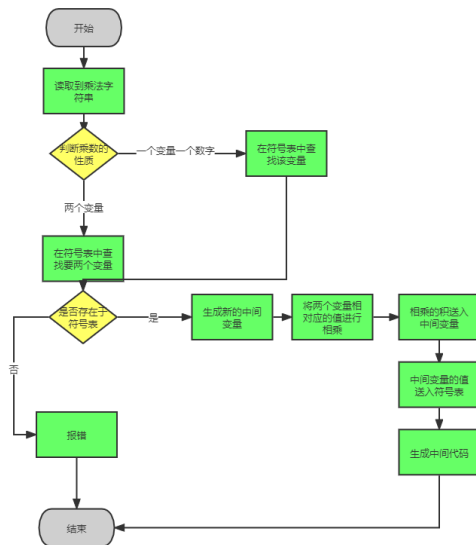
加法模块：



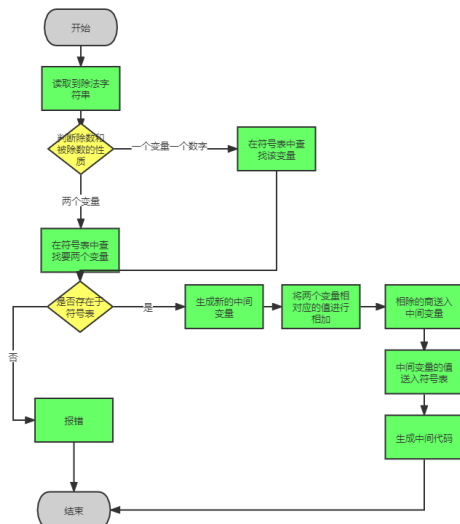
减法模块：



乘法模块：



除法模块:



四、函数功能:

词法分析器:

函数定义:

```

bool IsLetter(char ch); //判断是否为字母
bool IsDigit(char ch); //判断是否为数字
vector<string> scanner(string str); //扫描器, 进行词法分析
vector<string> scanner_1(string str); //修改的扫描器, 进行的词法分析用于中间代码生成
  
```

语法分析器:

变量定义:

非终结符定义

```
enum non_termin
{
    Program, SubProgram, TempProgram, Declaration, Type,
    TempBound, Function, Parameter, TempInt, RightEnd_1,
    IntParameter, Block, NumDeclaration, IntParameter_1, TempSentence,
    TempSentence_1, TempReturn, TempSentence_3, ReturnSentence, ReturnValue,
    WhileSentence, IfSentence, TempSentence_2, Expression, Relop,
    TempExpression, Token, TempSentence_4, Token_1, Token_2,
    Token_3, Call, TempSentence_5, TempExpression_1, RightEnd,
    Extra_1, TempLeftParenthesis, ParameterNum, SentenceEnd, TempFunction,
    Extra_2, JumpElse, Extra_3, Extra_4, Extra_5,
    Extra_6, Extra_7, Extra_8, Extra_9, Extra_10,
    Extra_11, Extra_12
};
string non_termin_string[NON_TERM_NUM] =
{
    "Program", "SubProgram", "TempProgram", "Declaration", "Type",
    "TempBound", "Function", "Parameter", "TempInt", "RightEnd_1",
    "IntParameter", "Block", "NumDeclaration", "IntParameter_1", "TempSentence",
    "TempSentence_1", "TempReturn", "TempSentence_3", "ReturnSentence", "ReturnValue",
    "WhileSentence", "IfSentence", "TempSentence_2", "Expression", "Relop",
    "TempExpression", "Token", "TempSentence_4", "Token_1", "Token_2",
    "Token_3", "Call", "TempSentence_5", "TempExpression_1", "RightEnd",
    "Extra_1", "TempLeftParenthesis", "ParameterNum", "SentenceEnd", "TempFunction",
    "Extra_2", "JumpElse", "Extra_3", "Extra_4", "Extra_5",
    "Extra_6", "Extra_7", "Extra_8", "Extra_9", "Extra_10",
    "Extra_11", "Extra_12"
};
```

终结符定义

```
enum termin
{
    _int, _void, _else, _if, _while,
    _return, _ID, _NUM, _assign, _plus,
    _minus, _multiply, _divide, _lower, _lower_equal,
    _larger, _larger_equal, _equal, _unequal,
    _bound, _comma, _left_parenthesis, _right_parenthesis,
    _left_brace, _right_brace, _end
};
string termin_string[TERM_NUM] = {
    "int", "void", "else", "if", "while",
    "return", "$ID", "$NUM", "=", "+",
    "-", "*", "/", "<", "<=",
    ">", ">=", "==", "!=",
    ";", ":", "(", ")",
    "{", "}", "#"
};
```

产生式定义：

```
vector<vector<int>> produce_array =
{
    {},
    {Program, SubProgram}, //1
    {SubProgram, Declaration, TempProgram}, //2
    {TempProgram, SubProgram}, //3
    {Declaration, int + NON_TERM_NUM, _ID + NON_TERM_NUM, Type}, //4
    {Declaration, _void + NON_TERM_NUM, _ID + NON_TERM_NUM, Function}, //5
    {Type, Extra_4, TempBound}, //6
    {Type, Function}, //7
    {TempBound, _bound + NON_TERM_NUM}, //8
    {Function, TempLeftParenthesis, _left_parenthesis + NON_TERM_NUM, Parameter, ParameterNum, _right_parenthesis + NON_TERM_NUM, Block, TempFunction}, //9
    {Parameter, TempInt}, //10
    {Parameter, _void + NON_TERM_NUM}, //11
    {TempInt, IntParameter, RightEnd_1}, //12
    {RightEnd_1, _comma + NON_TERM_NUM, TempInt}, //13
    {IntParameter, int + NON_TERM_NUM, _ID + NON_TERM_NUM, Extra_4}, //14
    {Block, _left_brace + NON_TERM_NUM, NumDeclaration, TempSentence, _right_brace + NON_TERM_NUM}, //15
    {NumDeclaration, IntParameter_1, Extra_4, _bound + NON_TERM_NUM, NumDeclaration}, //16
    {IntParameter_1, int + NON_TERM_NUM, _ID + NON_TERM_NUM}, //17
    {TempSentence, TempReturn, TempSentence_1}, //18
    {TempSentence_1, TempSentence}, //19
    {TempReturn, IfSentence}, //20
    {TempReturn, WhileSentence}, //21
    {TempReturn, ReturnSentence}, //22
    {TempReturn, TempSentence_3}, //23
    {TempSentence_3, Extra_5, Extra_9, _ID + NON_TERM_NUM, Extra_9, _assign + NON_TERM_NUM, Expression, Extra_6, Extra_12, _bound + NON_TERM_NUM}, //24
    {ReturnSentence, _return + NON_TERM_NUM, ReturnValue, SentenceEnd, _bound + NON_TERM_NUM}, //25
    {ReturnValue, Expression}, //26
    {WhileSentence, _while + NON_TERM_NUM, Extra_1, _left_parenthesis + NON_TERM_NUM, Extra_5, Expression, Extra_7, _right_parenthesis + NON_TERM_NUM, Block, Extra_8}, //27
}
```

函数定义：

```
void init_predict_table(); //对预测表进行初始化
void print_produce_expression(); //打印文法产生式
int find_non_termin_index(string expression); //查找非终结符的索引
int find_termin_index(string token); //查找终结符的索引
bool print_process(vector<string> input_string); //进行归约过程并将结果打印
```

```

{IfSentence_if + NON_TERM_NUM,Extra_1,_left_parenthesis + NON_TERM_NUM,Extra_5,Expression,Extra_7,_right_parenthesis + NON_TERM_NUM,Block,Extra_2,TempSentence_2,Extra_3},//28
{TempSentence_2,JumpElse_else + NON_TERM_NUM,Extra_1,Block,Extra_2},//29
{Expression,TempExpression,Relop},//30
{Relop,Extra_9,_lower + NON_TERM_NUM,Expression},//31
{Relop,Extra_9,_lower_equal + NON_TERM_NUM,Expression},//32
{Relop,Extra_9,_larger + NON_TERM_NUM,Expression},//33
{Relop,Extra_9,_larger_equal + NON_TERM_NUM,Expression},//34
{Relop,Extra_9,_equal + NON_TERM_NUM,Expression},//35
{Relop,Extra_9,_unequal + NON_TERM_NUM,Expression},//36
{TempExpression,TempSentence_4,Token},//37
{Token,Extra_9,_plus + NON_TERM_NUM,TempExpression,Extra_6},//38
{Token,Extra_9,_minus + NON_TERM_NUM,TempExpression,Extra_6},//39
{TempSentence_4,Token_2,Token_1},//40
{Token_1,Extra_9,_multiply + NON_TERM_NUM,TempSentence_4,Extra_6},//41
{Token_1,Extra_9,_divide + NON_TERM_NUM,TempSentence_4,Extra_6},//42
{Token_2,Extra_9,_NUM + NON_TERM_NUM},//43
{Token_2,_left_parenthesis + NON_TERM_NUM,Expression,_right_parenthesis + NON_TERM_NUM},//44
{Token_2,Extra_9,_ID + NON_TERM_NUM,Token_3},//45
{Token_3,Call},//46
{Call,Extra_10,_left_parenthesis + NON_TERM_NUM,TempExpression_1,Extra_11,_right_parenthesis + NON_TERM_NUM},//47
{TempSentence_5,TempExpression_1},//48
{TempExpression_1,Expression,RightEnd},//49
{RightEnd,Extra_11,_comma + NON_TERM_NUM,TempExpression_1},//50

```

预测表定义：

```

void init_predict_table()
{
    memset(table, -1, NON_TERM_NUM*TERM_NUM * sizeof(int));
    //将预测分析表中有状态转移的部分输入
    //Program
    table[Program][_int] = table[Program][_void] = 1;
    //SubProgram
    table[SubProgram][_int] = table[SubProgram][_void] = 2;
    //TempProgram
    table[TempProgram][_int] = table[TempProgram][_void] = 3;
    table[TempProgram][_end] = 0;
    //Declaration
    table[Declaration][_int] = 4;
    table[Declaration][_void] = 5;
    //Type
    table[Type][_bound] = 6;
    table[Type][_left_parenthesis] = 7;
    //TempBound
    table[TempBound][_bound] = 8;
    //Function
    table[Function][_left_parenthesis] = 9;
    //Parameter
    table[Parameter][_int] = 10;
    table[Parameter][_void] = 11;
    table[Parameter][_right_parenthesis] = 0;
    //TempInt

    //TempInt
    table[TempInt][_int] = 12;
    //RightEnd_1
    table[RightEnd_1][_comma] = 13;
    table[RightEnd_1][_right_parenthesis] = 0;
    //IntParameter
    table[IntParameter][_int] = 14;
    //Block
    table[Block][_left_brace] = 15;
    //NumDeclaration
    table[NumDeclaration][_int] = 16;
    table[NumDeclaration][_if] = table[NumDeclaration][_while] = table[NumDeclaration][_return] = table[NumDeclaration][_ID] = 0;
    //IntParameter_1
    table[IntParameter_1][_int] = 17;
    //TempSentence
    table[TempSentence][_if] = table[TempSentence][_while] = table[TempSentence][_return] = table[TempSentence][_ID] = 18;
    //TempSentence_1
    table[TempSentence_1][_if] = table[TempSentence_1][_while] = table[TempSentence_1][_return] = table[TempSentence_1][_ID] = 19;
    table[TempSentence_1][_right_brace] = 0;
    //TempReturn
    table[TempReturn][_if] = 20;
    table[TempReturn][_while] = 21;
    table[TempReturn][_return] = 22;
    table[TempReturn][_ID] = 23;
    //TempSentence_3
    table[TempSentence_3][_ID] = 24;
    //ReturnSentence
    table[ReturnSentence][_return] = 25;
}

```

```

//ReturnValue
table[ReturnValue][_ID] = table[ReturnValue][_NUM] = table[ReturnValue][_left_parenthesis] = 26;
table[ReturnValue][_bound] = 0;
//WhileSentence
table[WhileSentence][_while] = 27;
//IfSentence
table[IfSentence][_if] = 28;
//TempSentence_2
table[TempSentence_2][_else] = 29;
table[TempSentence_2][_if] = table[TempSentence_2][_while] = table[TempSentence_2][_return] = table[TempSentence_2][_ID] = table[TempSentence_2][_right_brace] = 0;
//Expression
table[Expression][_ID] = table[Expression][_NUM] = table[Expression][_left_parenthesis] = 30;
//Relop
table[Relop][_lower] = 31;
table[Relop][_lower_equal] = 32;
table[Relop][_larger] = 33;
table[Relop][_larger_equal] = 34;
table[Relop][_equal] = 35;
table[Relop][_unequal] = 36;
table[Relop][_bound] = table[Relop][_comma] = table[Relop][_right_parenthesis] = 0;
//TempExpression
table[TempExpression][_ID] = table[TempExpression][_NUM] = table[TempExpression][_left_parenthesis] = 37;
//Token
table[Token][_plus] = 38;
table[Token][_minus] = 39;
table[Token][_lower] = table[Token][_lower_equal] = table[Token][_larger] = table[Token][_larger_equal] = table[Token][_equal] = table[Token][_unequal] = table[Token][_bound] = table[Token][_comma];
//TempSentence_4
table[TempSentence_4][_ID] = table[TempSentence_4][_NUM] = table[TempSentence_4][_left_parenthesis] = 40;
//Token_1
table[Token_1][_multiplication] = 41;

```

产生式打印：

```

void print_produce_expression()
{
    cout << "文法产生式:" << endl;
    for (int i = 0; i < produce_array.size(); i++)
    {
        for (int j = 0; j < produce_array[i].size(); j++)
        {
            if (produce_array[i][j] < NON_TERM_NUM)
                cout << non_termin_string[produce_array[i][j]] << " ";
            else
                cout << termin_string[produce_array[i][j] % NON_TERM_NUM] << " ";
            if (j == 0)
            {
                cout << "-> ";
            }
        }
        cout << endl;
    }
}

```

归约过程：

```

void print_process(vector<string> input_string)
{
    vector<string> analyze_stack;
    analyze_stack.push_back("#");
    analyze_stack.push_back(non_termin_string[Program]);
    input_string.push_back("#");
    int count = 0;
    while (!analyze_stack.empty())
    {
        count++;
        cout << "第" << count << "步: " << endl;
        cout << "语法栈:";
        for (int i = 0; i < analyze_stack.size(); i++)
        {
            cout << analyze_stack[i] << " ";
        }
        cout << endl;
        cout << "输入串:";
        for (int i = 0; i < input_string.size(); i++)
        {
            cout << input_string[i] << " ";
        }
        cout << endl << endl;
        if (analyze_stack[analyze_stack.size() - 1] != input_string[0]) //若分析栈的栈顶和输入串的栈顶不同
        {
            //找出在预测分析表中的入口位置
            int non_termin_index = find_non_termin_index(analyze_stack[analyze_stack.size() - 1]);
            int termin_index = find_termin_index(input_string[0]);
            //预测表中的产生式编号
            int produce_index = table[non_termin_index][termin_index];
            //进行归约操作
        }
    }
}

```



```

    if (produce_index == -1)
    {
        cout << "error!" << endl;
        return;
    }
    else if (produce_index == 0)
    {
        analyze_stack.pop_back();
    }
    else
    {
        analyze_stack.pop_back();
        for (int i = produce_array[produce_index].size() - 1; i > 0; i--)
        {
            if (produce_array[produce_index][i] < NON_TERM_NUM)
            {
                analyze_stack.push_back(non_termin_string[produce_array[produce_index][i]]);
            }
            else
            {
                analyze_stack.push_back(termin_string[produce_array[produce_index][i] % NON_TERM_NUM]);
            }
        }
    }
}
else
{
    analyze_stack.pop_back();
    input_string.erase(input_string.begin());
}
}
}

```

中间代码生成器：

函数定义：

```

void semantic(vector<string> result); //控制器，调用工具函数
bool is_exist_in_table(string par); //判断符号是否在符号表中
void definition(vector<string> &test); //变量定义语义分析
bool assign(vector<string> &test); //变量赋值语义分析
bool _plus(vector<string> &test); //加法语义分析
bool _multiply(vector<string> &test); //乘法语义分析
bool _minus(vector<string> &test); //减法语义分析
bool _divide(vector<string> &test); //除法语义分析

int get_par_index(string par); //获取变量在符号表中的入口
void print_test(vector<string> test); //打印中间代码
void print_par(vector<string> par_name, vector<int> par_value); //打印变量符号表，测试用
bool par_is_letter(string par); //区分变量和数字
vector<string> strip(vector<string> &test); //分解字符串，以语句为单位分析
bool if_sentence(vector<string> &test); //if语义分析
bool while_sentence(vector<string> &test); //while语义分析
bool return_sentence(vector<string> &test); //return语义分析
bool else_sentence(vector<string> &test); //else语义分析

```

四、运行结果

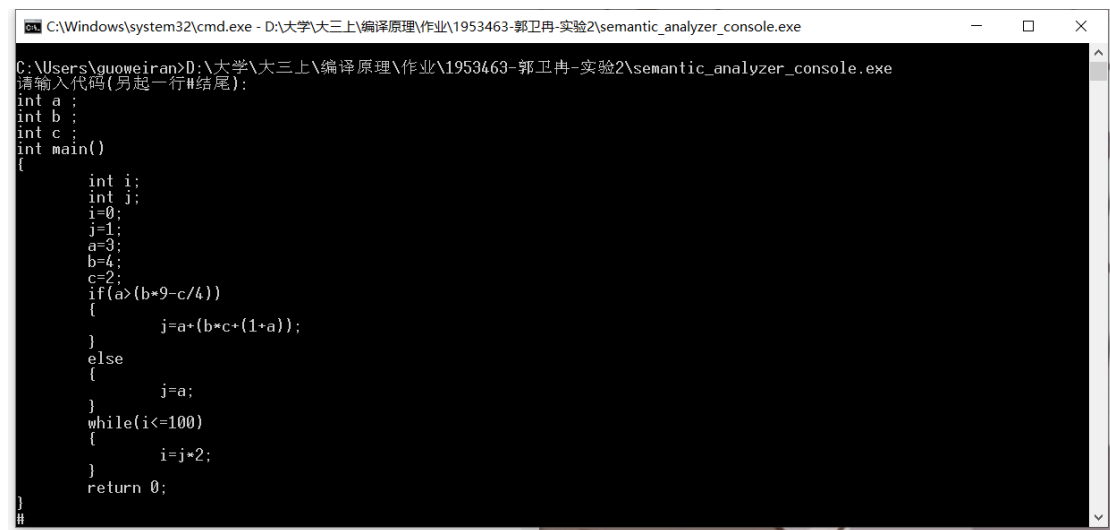
本程序使用测试用例 test.txt、test_wrong1.txt 和 test_wrong2，分别对应一个正确测试用例和两个错误测试用例。

①无图形界面

没有使用图形界面，打开可执行文件（作业目录下的 semantic_analyzer_console.exe），可以看见输入代码的提示。



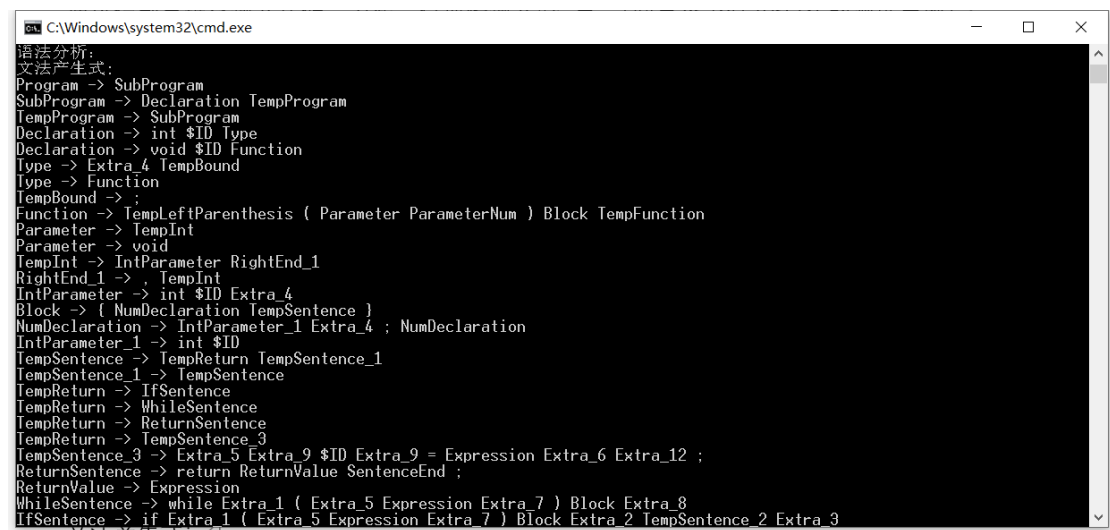
通过控制台键盘输入代码，另起一行顶格输入#结束。（同目录下有 test.txt 的测试用例）



输出结果如下所示：

文法产生式：

如下图所示，是经过将文法转化为左递归文法的形式，打印在屏幕上。



归约过程：

如下图所示，输出语法栈和输入串中的的内容，并一步步打印归约步骤，每次归约的步骤数都会标出。

```
C:\Windows\system32\cmd.exe
第1步:
语法栈:# Program
输入串:int $ID ; int $ID ; int $ID ; int $ID ( ) { int $ID ; int $ID ; $ID = $NUM ; $ID = $NUM ; $ID = $NUM ; $ID = $NUM
; $ID = $NUM ; if ( $ID > ( $ID * $NUM - $ID / $NUM ) ) { $ID = $ID + ( $ID * $ID + ( $NUM + $ID ) ) ; } else { $ID = $
ID ; } while ( $ID <= $NUM ) { $ID = $ID * $NUM ; } return $NUM ; } #

第2步:
语法栈:# SubProgram
输入串:int $ID ; int $ID ; int $ID ; int $ID ( ) { int $ID ; int $ID ; $ID = $NUM ; $ID = $NUM ; $ID = $NUM ; $ID = $NUM
; $ID = $NUM ; if ( $ID > ( $ID * $NUM - $ID / $NUM ) ) { $ID = $ID + ( $ID * $ID + ( $NUM + $ID ) ) ; } else { $ID = $
ID ; } while ( $ID <= $NUM ) { $ID = $ID * $NUM ; } return $NUM ; } #

第3步:
语法栈:# TempProgram Declaration
输入串:int $ID ; int $ID ; int $ID ; int $ID ( ) { int $ID ; int $ID ; $ID = $NUM ; $ID = $NUM ; $ID = $NUM ; $ID = $NUM
; $ID = $NUM ; if ( $ID > ( $ID * $NUM - $ID / $NUM ) ) { $ID = $ID + ( $ID * $ID + ( $NUM + $ID ) ) ; } else { $ID = $
ID ; } while ( $ID <= $NUM ) { $ID = $ID * $NUM ; } return $NUM ; } #

第4步:
语法栈:# TempProgram Type $ID int
输入串:int $ID ; int $ID ; int $ID ; int $ID ( ) { int $ID ; int $ID ; $ID = $NUM ; $ID = $NUM ; $ID = $NUM ; $ID = $NUM
; $ID = $NUM ; if ( $ID > ( $ID * $NUM - $ID / $NUM ) ) { $ID = $ID + ( $ID * $ID + ( $NUM + $ID ) ) ; } else { $ID = $
ID ; } while ( $ID <= $NUM ) { $ID = $ID * $NUM ; } return $NUM ; } #

第5步:
语法栈:# TempProgram Type $ID
输入串:$ID ; int $ID ; int $ID ; int $ID ( ) { int $ID ; int $ID ; $ID = $NUM ; $ID = $NUM ; $ID = $NUM ; $ID = $NUM ; $
ID = $NUM ; if ( $ID > ( $ID * $NUM - $ID / $NUM ) ) { $ID = $ID + ( $ID * $ID + ( $NUM + $ID ) ) ; } else { $ID = $ID ;
} while ( $ID <= $NUM ) { $ID = $ID * $NUM ; } return $NUM ; } #
```

中间代码:

如下图所示, 生成了中间代码。

```
C:\Windows\system32\cmd.exe
第432步:
语法栈:#
输入串:#

中间代码生成:
0:(=,0,_,1)
1:(=,1,_,1)
2:(=,3,_,a)
3:(=,4,_,b)
4:(=,2,_,c)
5:(=,b,9,10)
6:(/,c,4,11)
7:(=,10,11,12)
8:(j>,a,12,14)
9:(=,b,c,13)
10:(=,a,1,14)
11:(=,T3,T4,15)
12:(=,a,15,16)
13:(=,16,_,1)
14:(j,_,_,16)
15:(=,a,_,j)
16:(j<=,1,100,18)
17:(j,_,_,21)
18:(=,j,2,17)
19:(=,17,_,1)
20:(j,_,_,16)
21:
C:\Users\guoweiran>
```

错误处理:

若在某个步骤发生了错误, 则在该步骤报错, 并终止整个程序。

```
C:\Windows\system32\cmd.exe
语法栈:# TempProgram TempFunction } TempSentence_1 ; SentenceEnd Relop Token Token_1 $NUM
输入串:$NUM ; #

第423步:
语法栈:# TempProgram TempFunction } TempSentence_1 ; SentenceEnd Relop Token Token_1
输入串:; #

第424步:
语法栈:# TempProgram TempFunction } TempSentence_1 ; SentenceEnd Relop Token
输入串:; #

第425步:
语法栈:# TempProgram TempFunction } TempSentence_1 ; SentenceEnd Relop
输入串:; #

第426步:
语法栈:# TempProgram TempFunction } TempSentence_1 ; SentenceEnd
输入串:; #

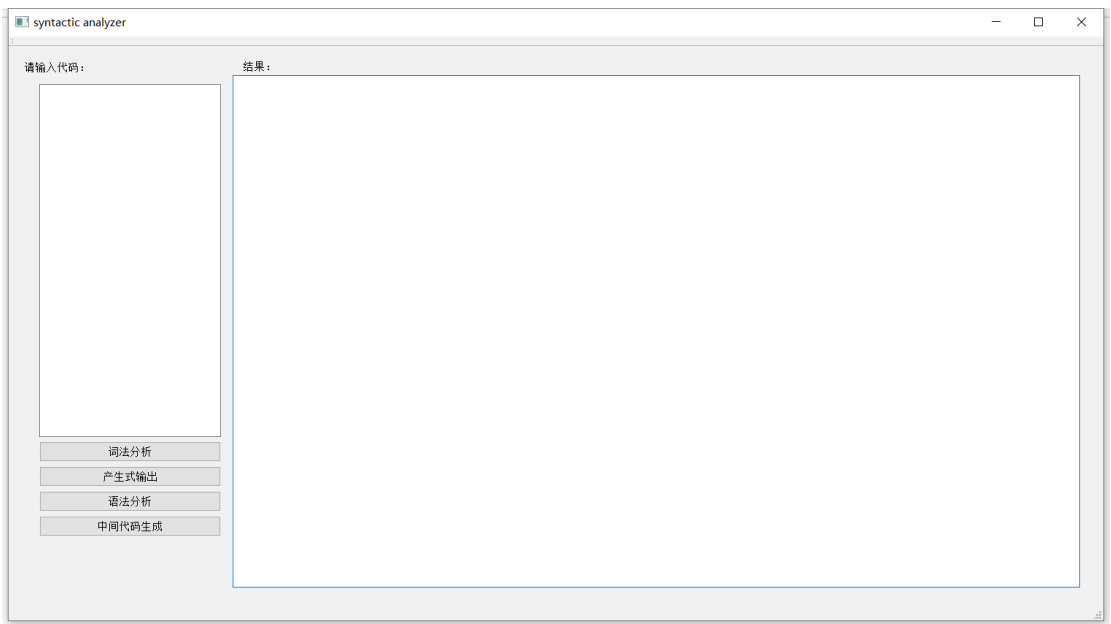
第427步:
语法栈:# TempProgram TempFunction } TempSentence_1 ;
输入串:; #

第428步:
语法栈:# TempProgram TempFunction } TempSentence_1
输入串:#

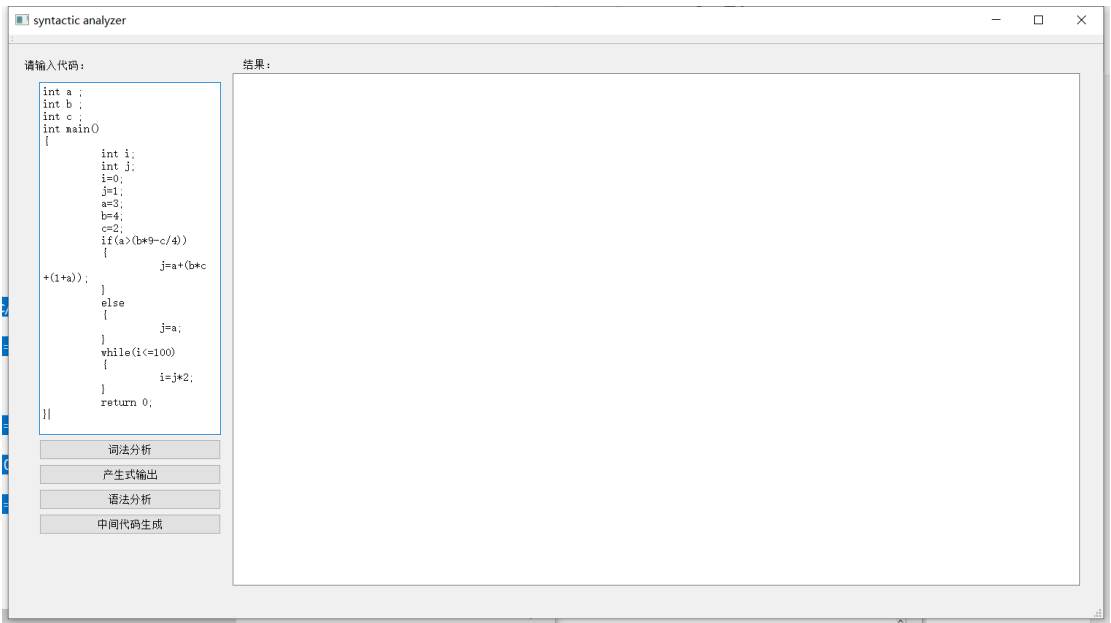
error!
C:\Users\guoweiran>
```

②图形化界面

打开可执行文件 semantic_anlayzer_gui（作业目录下的 semantic_analyzer_gui/semantic_analyzer_gui.exe），可运行图形化的词法分析器。
图形化界面如图所示
左半部分输入代码，右半部分输出结果。

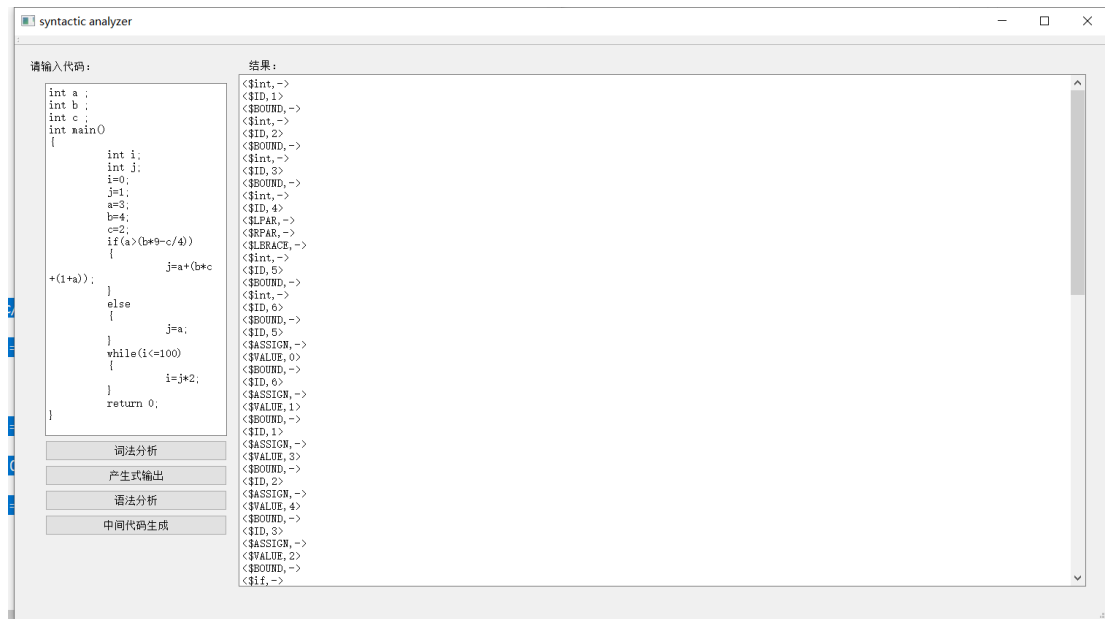


输入代码：



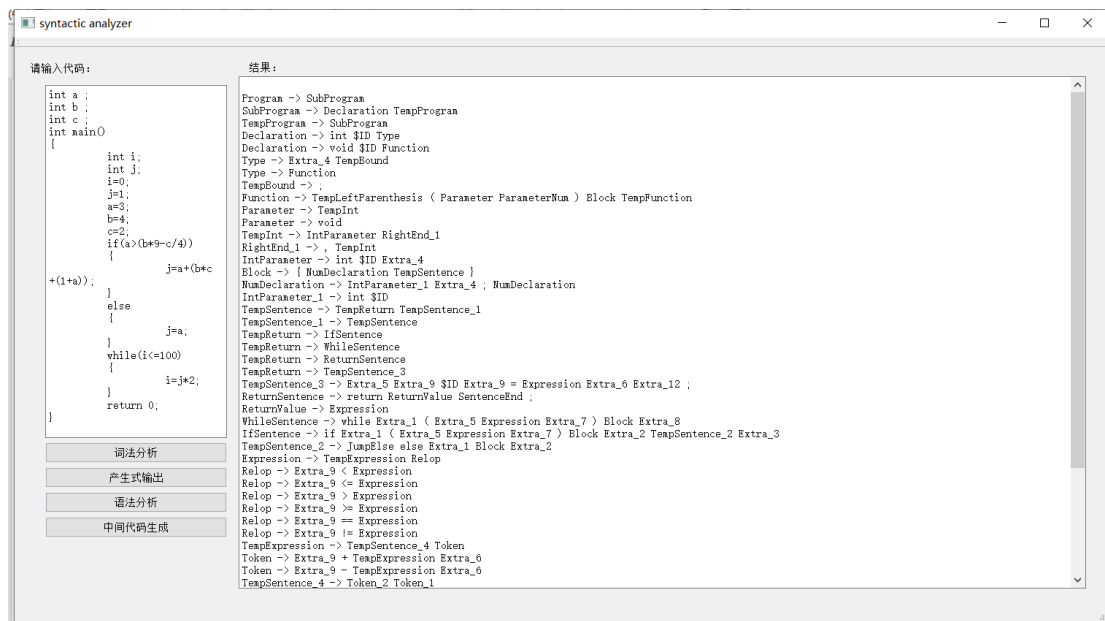
词法分析

按下“词法分析”的按钮，右方输出词法分析结果。

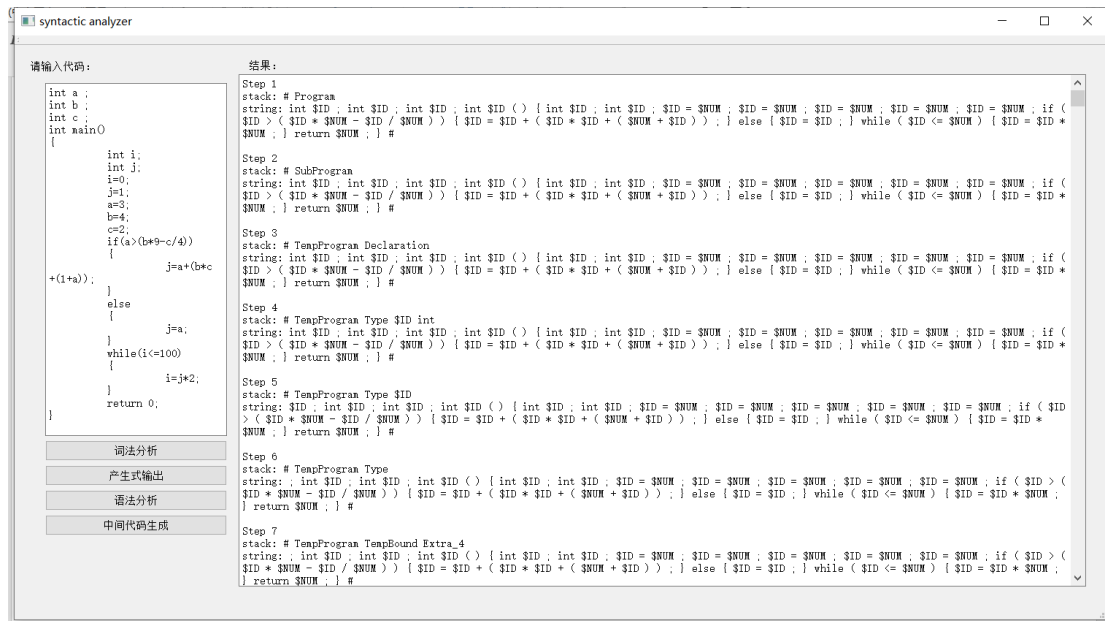


语法分析

按下“产生式输出”的按钮，右方输出产生式。



按下“语法分析”的按钮，右方输出语法分析。

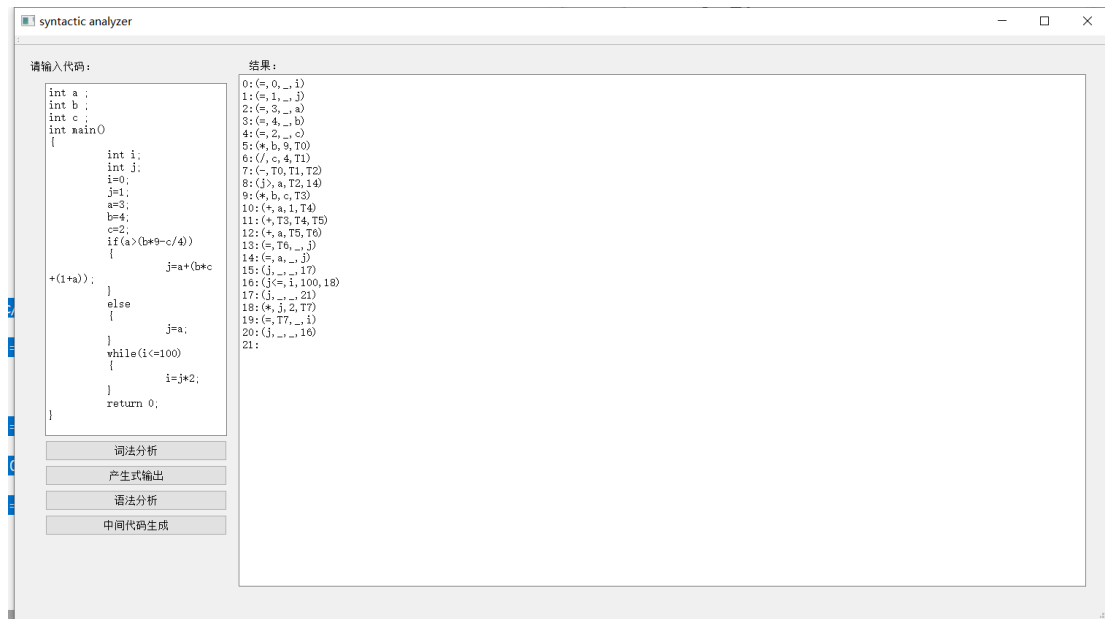


语法分析打印内容：步骤、语法栈和输入串中内容。

通过右侧输出的结果，可以清楚看到是哪里出现了错误。

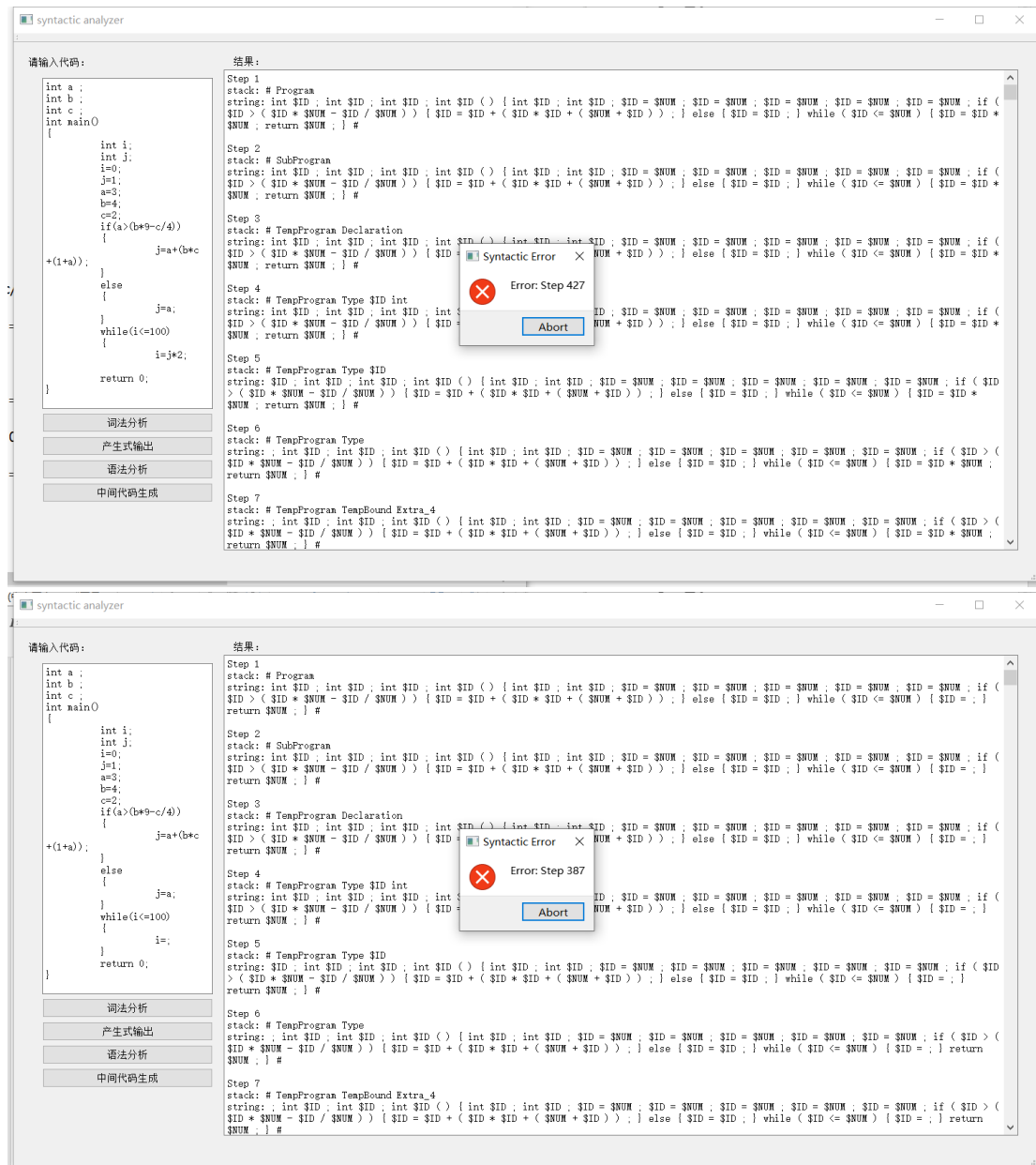
中间代码生成器：

点击“中间代码生成”按钮，右方输出生成的中间代码。

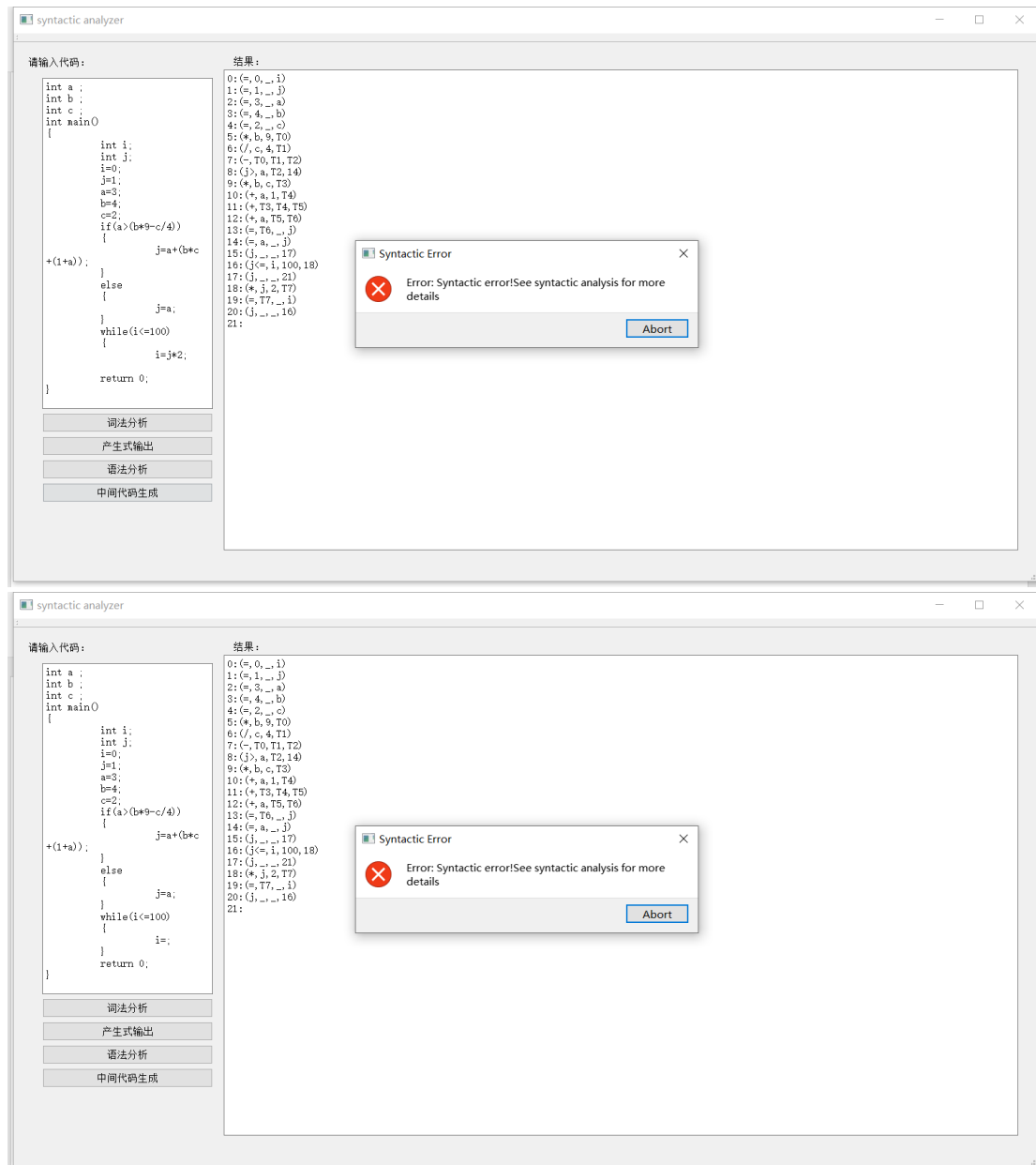


出错处理：

若语法分析过程中出现错误，则弹出错误窗口，并指出错误的步骤数。（压缩包中有 test_wrong_1.txt 和 test_wrong_2.txt 的测试用例）



若在中间代码生成出现的错误，也会有弹窗显示提示，并指出错误的原因。



五、总结与感想

考虑到更为通行的高级语言语义检查和中间代码生成，需要考虑的内容有以下几点：

①变量、数组等的语义分析和地址组织

解决方法：在语义分析时解析出数组存储所需的要素，即变量类型、变量名、变量的值，如果是数组还需要分析出数组的大小，在读或者写数组元素的时候，需要根据索引得出偏移量。数组中的元素连续存放。

②调用函数时地址的跳转

解决方法：利用运行栈，存储当前函数中的具体信息，如形式参数、变量、返回地址等内容，每调用一次函数就压入一次运行栈，运行栈之间用调用地址和返回地址相连接。

③表达式的解析

解决方法：由于表达式是由加减乘除法所构成的式子，运算顺序和括号的存在可能会导致表达式的解析更加复杂。通过构造运算树的方式，按照运算顺序依次计算表达式的值。