同济大学计算机系

编译原理课程设计实验报告

学　　号　2152118

姓　　名　史君宝

专　　业　计算机科学与技术

授课老师　丁志军

一、需求分析

1. 实验环境

操作系统：Windows11 专业中文版

软件环境：Qt 5.14.2

编码语言：C++语言

2. 实验内容

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。

3. 程序任务输入及其范围

在本次实验中我们会给出相应的源程序（C 语言程序），还有词法文法和语法文法。我们按照实验指导书上的进行举例：

（1）C 程序：

```
int program(int a,int b,int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
    {
        i=j*2;
    }
    return i;
}
```

（2）词法文法：

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

（3）语法文法：

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>
<变量声明> ::= ;
<函数声明> ::= '('<形参>')' <语句块>
<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>;
<return 语句> ::= return [<表达式>] ;**（注：[ ]中的项表示可选）**

4. 输出形式

在本次设计中，我们完成了一定的 C 语言编译器的功能，能够通过读取词法文法和语法文法之后，对 C 程序进行分析，输出一定的错误分析或者最终的目标代码。

错误分析：

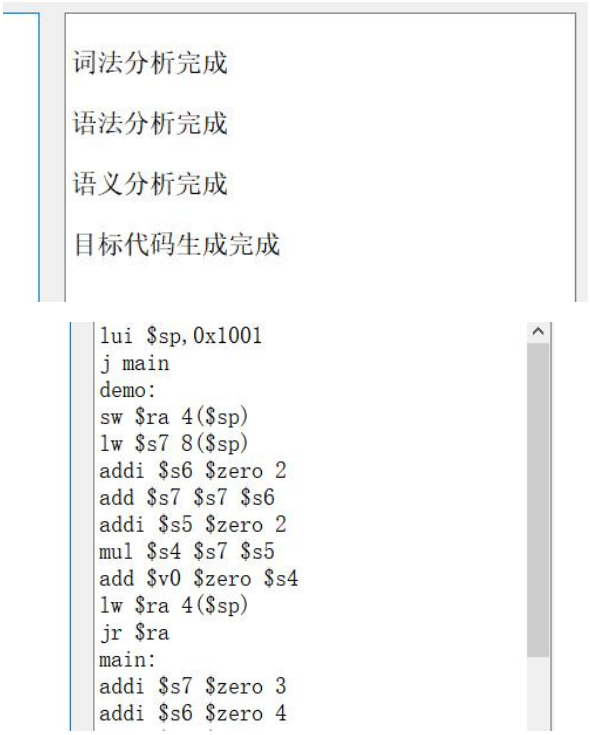
提示栏

error: 分析失败，程序存在语法错误

语义分析完成

目标代码生成完成

正确结果：



5. 程序功能：

完成了一定的C语言编译器的功能，能够通过读取词法文法和语法文法之后，对C程序进行分析，输出一定的错误分析或者最终的目标代码。

6. 测试数据：

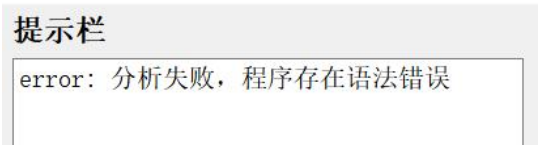
正确：

```
int a;
int b;
int program(int a,int b,int c)
{
	int i;
	int j;
	i=0;
	if(a>(b+c))
	{
		j=a+(b*c+1);
	}
	else
	{
		j=a;
	}
	while(i<=100)
	{
		i=j*2;
		j=i;
	}
	return i;
}

int demo(int a)
{
	a=a+2;
	return a*2;
}
```


```
lui $sp,0x1001
j main
demo:
sw $ra 4($sp)
lw $s7 8($sp)
addi $s6 $zero 2
add $s7 $s7 $s6
addi $s5 $zero 2
mul $s4 $s7 $s5
add $v0 $zero $s4
lw $ra 4($sp)
jr $ra
main:
addi $s7 $zero 3
addi $s6 $zero 4
```

错误：

```
int max(int x,int y){
    if(x>y)
        return x;
    else
        return y;
}
int min(int x,int y){
    if(x<y){
        int t;
        t=x;
        x=y;
        y=t;
    }
    return y;
}
int sum(int x,int y){
    return x+y;
}
int main()
{
    int a;
    int b;
    int c;
    int d;
    int f;
    a=3;
    b=4;
    c=max(a,b);
    d=sum(a,b);
    f=min(c,d);
```




```
int max(int x,int y){
    if(x>y){
        return x;
    }
    else{
        return y;
    }
}
int min(int x,int y){
    if(x<y){
        int t;
        t=x;
        x=y;
        y=t;
    }
    return y;
}
int sum(int x,int y){
    return x+y;
}
```

提示栏

error: 分析失败，程序存在语法错误

```
int max(int x,int y){
    if(x>y){
        return x;
    }
    else{
        return y;
    }
}
int min(int x,int y){
    if(x<y){
        int t;
        t=x;
        x=y;
        y=t;
    }
    return y;
}
int sum(int x,int y){
    return x+y;
}
int main()
{
    int a;
    int b;
    int c;
    int d;
    a=3;
    b=4;
    c=max(a,b);
    d=sum(a,b);
```

提示栏

error: 语法错误：第31行，变量f未声明

二、概要设计

1.任务分解：

（1）词法分析：

<1>初始化词法分析器，包括定义关键字、运算符、界符等的词法规则。

<2>从源文件中逐个字符读取，并跳过空格和换行符。

<3>根据读入字符的类型确定词的类型，如标识符、关键字、常量等。

<4>使用有限自动机（DFA）或正则表达式进行词法分析。

<5>输出词法单元序列，每个词法单元包含词的值和类型。

（2）语法分析：

<1>读取给定的语法规则，构建语法分析器。

<2>根据语法规则生成语法分析表，如 LL(1)分析表或 LR 分析表。

<3>使用自顶向下或自底向上的方法语法分析，生成语法树或语法分析树。

<4>在语法分析过程中进行语义检查，如类型检查、作用域检查等。

<5>输出抽象语法树（AST）或中间代码表示。

（3）语义分析（Semantic Analysis）：

<1>进行语义分析，包括类型检查、作用域分析、常量折叠等。

<2>根据语义规则对 AST 进行遍历，执行语义动作。

<3>检测并纠正语法错误和不合理的代码结构。

<4>输出经过语义分析的中间表示形式。

（4）中间代码生成（Intermediate Code Generation）：

<1>将经过语法和语义分析的源代码转换为中间代码表示形式。

<2>生成中间代码，如三地址码、四地址码或类似的表示形式。

<3>对中间代码进行优化，如常量传播、死代码消除等。

（5）目标代码生成（Code Generation）：

<1>将优化后的中间代码转换为目标机器代码。

<2>选择合适的寄存器分配策略。

<3>生成目标代码，如汇编代码或机器代码。

<4>输出可在目标机器上执行的最终目标代码。

2. 数据类型的定义：

（1）词法分析器：

数据类型：

```
// 枚举数据类型
enum __DataType
{
    DINT,
    DVOID
};
```

申明语句类型：

```
// 枚举声明语句的类型(变量声明和函数声明)
enum __DeclareType
{
    FUN,
    VAR
};
```

操作符类型：

```
    // 运算符
    ADD,
    MINUS,
    MULT,
    DIV,
    ASSIGN,
    // 比较符
    GT,
    LT,
    GTE,
    LTE,
    EQU,
```

关键词类型：

```
    // 关键字
    INT,
    VOID,
    RETURN,
    WHILE,
    ELSE,
```

识别具体字符：

```
bool isLetter(char x);
bool isKeyWord(string str);
bool isBiOperator(string str);
bool isMoOperator(char x);
bool isDelimiter(char x);
bool isState(char x);
bool isFinalState(char x);
bool isExist(char x, NFAstateSet now);
void NFABuild(string grammarFileName);
int isExsitState(NFAstateSet now);
```

字符类型：

```
// 关键字 类型: string
const string keyWords[KEYWORD_NUMS] = {"break", "case", "char", "continue", "do", "default", "double",
                                       "else", "float", "for", "if", "int", "include", "long", "main", "return
// 双目运算符 类型: string
const string biOperator[BO_NUMS] = {"++", "--", "&&", "||", "<=", "!=", "==", ">=", "+=", "-=", "*=", "/="};
// 单目运算符 类型:  char
const char moOperator[MO_NUMS] = {'+', '-', '*', '/', '!', '%', '~', '&', '|', '^', '=', '>', '<'};
// 界符 类型: char
const char delimiter[DELIMITER_NUMS] = {',', '(', ')', '{', '}', ';', '#', '$'};
```

（2）语法分析器：

语法操作类型：

```
// 枚举操作类型
enum __Operator
{
    shift,  // 移进
    reduct, // 规约
    acc,    // 接受
    error   // 报错
};
```

语法分析相关函数：

```
string _grammarFileName;                                                    // 文法文件名称
vector<_Production> _productions;                                           // 产生式
map<_Symbol, set<_Symbol>> _firstSet;                                       // first集合
_Dfa _dfa;                                                                  // DFA
_ActionGoTo _actionGoTo;                                                    // LR1文法对应的actionGoTo表
stack<_Symbol *> _symStack;                                                 // 符号栈
stack<int> _staStack;                                                       // 状态栈
_TempVar _tempVar;                                                          // 中间临时变量
vector<_Var> _varTable;                                                     // 变量表
vector<_Fun> _funTable;                                                     // 函数表
_MediateCodeGen _code;                                                      // 中间代码
vector<pair<int, string>> fun_enter;                                        // 函数入口
_Var *_LookUpVar(string _varName);                                          // 查找变量
_Fun *_LookUpFunc(string _funName);                                         // 查找函数
int _nowLevel;                                                              // 当前层级
bool _FunParMatch(list<string> &_argumentList, list<__DataType> &_parameterList); // 判断函数参数是否匹配
```

（3）语义分析器：

四元组：

```
// 四元组
struct _Quaternary
{
    string _opt; // 操作符
    string _rs;  // rs 操作数
    string _rt;  // rt 操作数
    string _rd;  // rd操作数
};
```

语义分析代码：

```
void _GrammarParse::_SemanticAnalyse(list<Word> &_lexicalResult)
{
    // 在这里面进行自下而上的分析
    bool _acc = false; // 是否接受
    int _linesCnt = 1; // 记录行数
    // 初始化符号栈和状态栈
    _symStack.push(new _Symbol{"$", 1});
    _staStack.push(0);

    list<Word>::iterator _scanIter; // 扫描程序
    for (_scanIter = _lexicalResult.begin(); _scanIter != _lexicalResult.end();)
    {
        // 获得当前词的类型和符号
        auto _nowSymbol = _scanIter->first;
        auto _nowType = _scanIter->second;
        // cout << _nowSymbol << "\_n";
        if (_nowType == LCOMMENT || _nowType == SCOMMENT)
        {
            continue;
        }
```

（4）目标代码生成器：
基本块：

```
// 基本块
struct _BaseBlock
{
    string _blockName;              // 块名
    int _nextBlockA;                // 下一块连接块
    int _nextBlockB;                // 下一块连接块
    vector<_Quaternary> _quaCode; // 四元式代码
};
```

中间代码与基本块

```
class _MediateCodeGen
{
public:
    _Label _label;                                  // 下一个位置
    map<string, vector<_BaseBlock>> _funBlocks; // 函数块
    vector<_Quaternary> _quaCode;                // 四元式代码

    void _Emit(_Quaternary _quaternary);              // 将四元式插入到code中
    void _BackPatch(list<int> _nextList, int _quad); // 回填操作
    int _GetNextQuad();                               // 获取下一条语句的编号
};
```

```
// 带活跃和待用信息的四元式
class InfoQuaternary
{
public:
    _Quaternary base_q;
    Info info[3]; // 分别是src1 src2 des的信息情况
    InfoQuaternary(_Quaternary base_q, Info i1, Info i2, Info i3);
    InfoQuaternary(_Quaternary base_q, Info i[]);
    InfoQuaternary(_Quaternary base_q);
};
```

```
// 带活跃和待用信息的Block
struct InfoBlock
{
    string block_name;                          // 块名
    int nxtb[2];                                // 两个出口
    vector<InfoQuaternary> codes;               // 这里带活跃和待用信息
    InfoBlock();                                // 默认构造
    InfoBlock(struct BaseBlock &base_block); // 用baseblock转化
};
```

3. 主程序流程：

（1）先进入 main 函数调用开始界面函数，StartWindow 函数：

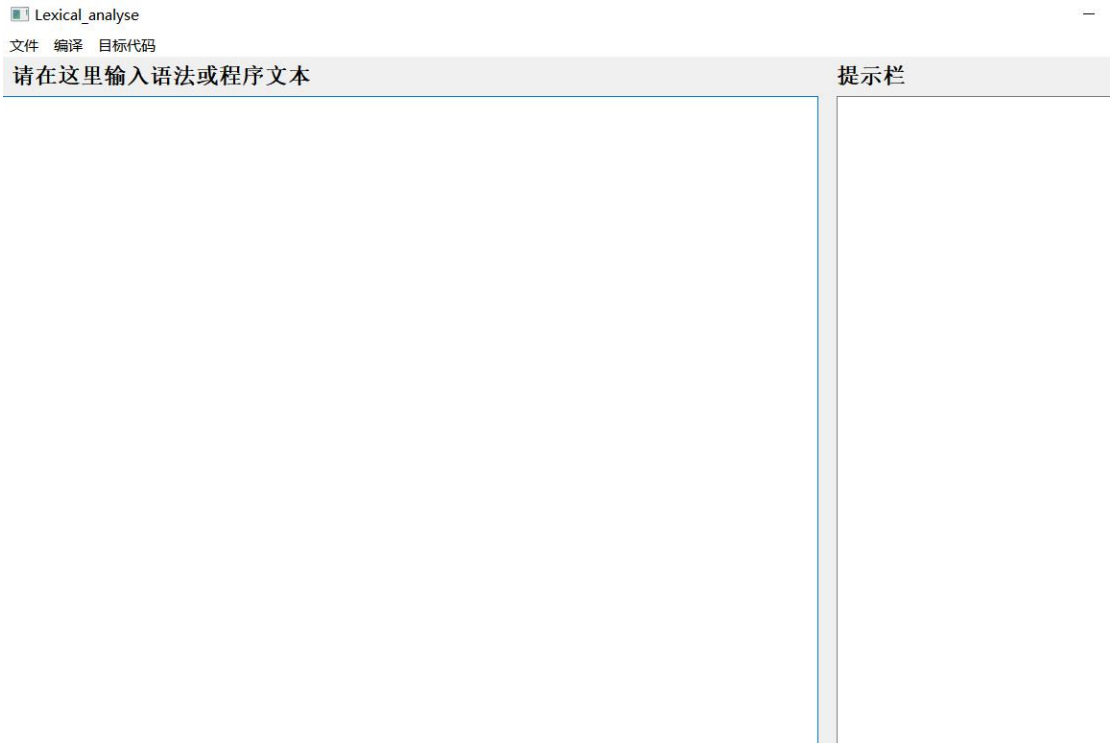
```
#include "mainwindow.h"
#include "StartWindow.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    StartWindow w;
    w.show();
    return a.exec();
}
```

（2）开始界面函数会连接主窗口：

```
#ifndef STARTWINDOW_H
#define STARTWINDOW_H

#include <QMainWindow>
#include "mainwindow.h"
class StartWindow : public QMainWindow
{
    Q_OBJECT

public:

    StartWindow();

    void paintEvent(QPaintEvent *);

    MainWindow * mainwindow = NULL;


signals:

};

#endif // STARTWINDOW_H
```

开始窗口展示：

（3）点击 start 之后会进入主窗口，主窗口是主要的交互界面。



4. 模块间的调用关系：

（1）导入源程序：

```cpp
//使用文件名导入源程序
void MainWindow::on_Open_file_triggered()
{
    bool ok;
    QString fileName = QInputDialog::getText(this, "输入文件名", "文件名:", QLineEdit::Normal, "", &ok);
    if (!ok || fileName.isEmpty()) {
        QMessageBox::warning(this, "警告", "未输入文件名");
        return;
    }

    fileName = "file/" + fileName;

    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
       QMessageBox::warning(this, "警告", "无法打开文件");
       return;
    }

    QTextStream in(&file);
    QString fileContent = in.readAll();
    file.close();
```

```cpp
//使用文本框导入源程序
void MainWindow::on_Open_text_triggered()
{
    QString text = this->ui->Grammar_Program_Textin->toPlainText();
    if (text.isEmpty()) {
        QMessageBox::warning(this, "Warning", "Program text is empty.");
    }
    else
    {
        ofstream fout(R"(file\Program.txt)");
        if (!fout.is_open())
        {
            QMessageBox::warning(nullptr, "警告", "无法打开程序文件");
            return;
        }

        fout << text.toStdString();
        fout.close();

        QMessageBox::information(this, "提示", "文本框中的内容已复制到 Program.txt 文件中");
        return;
    }
}
```

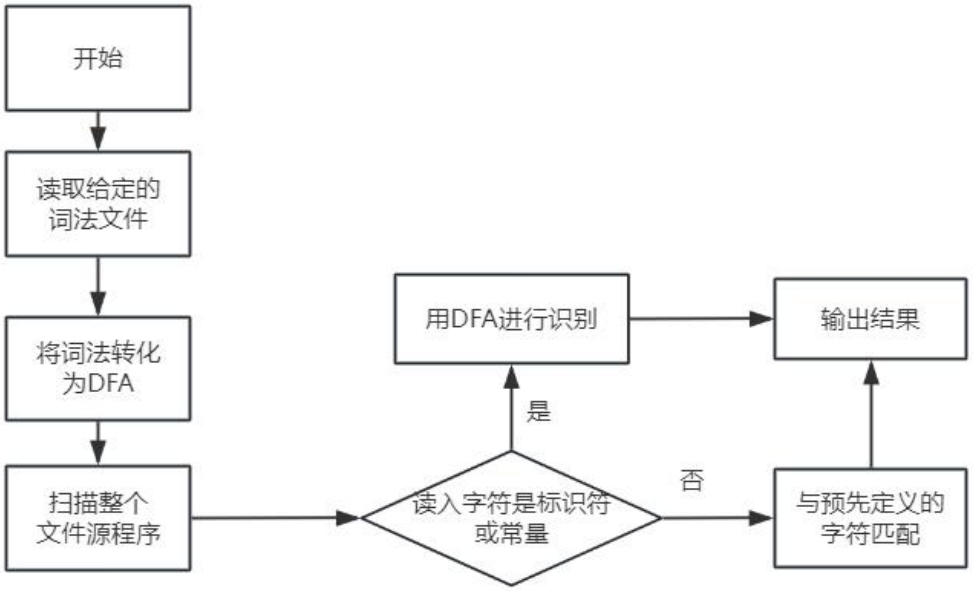
本设计中采用两种导入源程序的方式，分别是文本框导入和文件名导入，导入源程序之后可以进行进一步的操作。

（2）词法分析：

词法分析代码：

```cpp
// 先调词法工作
wordParse wp;
this->ui->Text_help->setText("");
wp.work(file_name);
wp.outputParseResult(R"(file\parseResult.txt)");
```

```cpp
void wordParse::work(string file_name)
{
    parseMapInit();
    initSet();
    NFABuild(R"(file\WordParseGrammar.txt)");
    NFAConvert();
    fileSource = fopen(file_name.c_str(), "r+");
    // outputFile.open("output.txt");
    // resultFile.open("ParseAnalyseResult.txt");
    scanFile();
    fclose(fileSource);
    // outputFile.close();
}
```

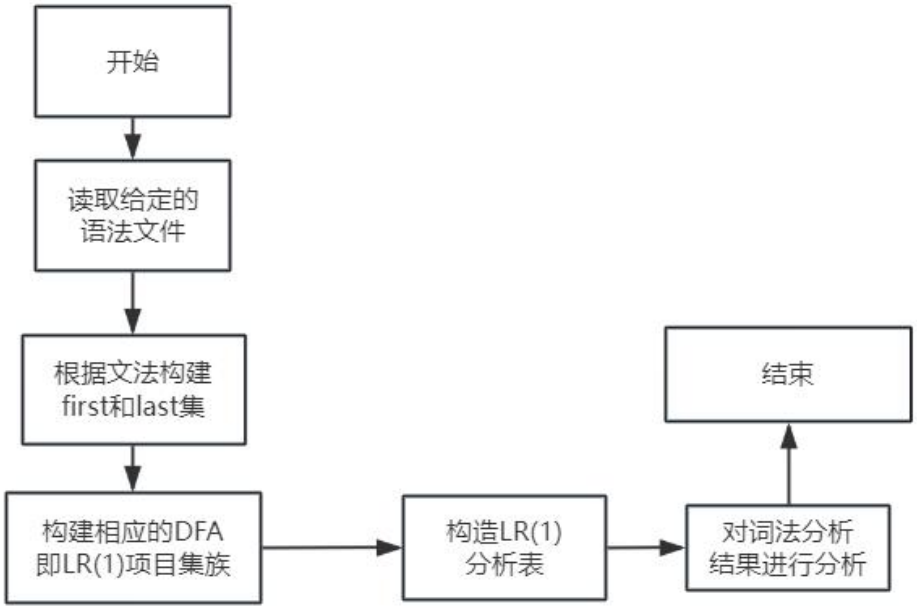


（3）语法分析：

语法分析代码：

```
_GrammarParse gp(R"(file\Grammar.txt)");
gp._Work();
```

```
void _GrammarParse::_Work()
{
    _ReadProductions();
    _GetFirstSet();
    _BuildDfa();
}
```



（4）语义分析：

语义分析代码：

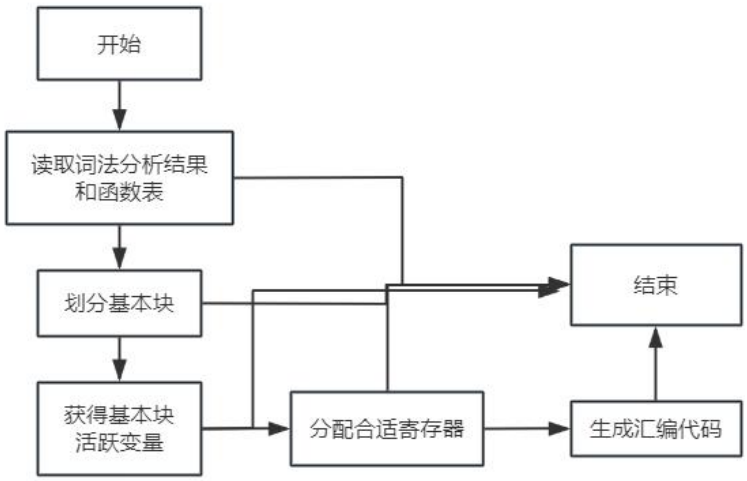
```
gp._SemanticAnalyse(wp.getWordParseResult());
```

（5）中间代码生成：

相关代码：

```
ObjectCodeGenerator ocg(gp._code._quaCode, gp.fun_enter);

ocg.generate_code();
```

```
void ObjectCodeGenerator::generate_code()
{
    // 1层循环 为整个中间代码生成目标代码
    object_codes.push_back("lui $sp,0x1001"); // 设置栈顶
    object_codes.push_back("j main");         // 跳转到main
    for (auto fun_iter = block_generator.info_fun_blocks.begin(); fun_iter != block_generator.info_fun_blocks.end(); fun_iter++)
    {
        generate_code_for_fun(fun_iter);
    }
    object_codes.push_back("end:");

    // 打印结果
    cout << "目标代码生成完毕！"
         << "\n";
    ofstream fout;
    fout.open(R"(file\code.asm)");
    for (auto &e : object_codes)
        fout << e << "\n";
    fout.close();
}
```



三、详细设计

1. 重点函数和重点变量

（1）词法分析：

重点函数和变量	功能
map<string, WordType> parseMap;	词法分析对照表
list<Word> parseResult;	词法分析结果
set<string> keyWordsSet;	关键字集合
set<string> biOperatorSet;	操作符集合
set<char> stateSet;	状态符集合
set<char> finalStateSet;	终结符集合
char DFAStartState;	构建 DFA 初始状态
NFAstateSet stateTransfer[STATE_NUMS][STATE_NUMS];	NFA 的状态集合
void **parseMapInit**();	词法分析表初始化
void **initSet**();	初始化各集合
void **parseMapInit**(); void **initSet**(); bool **isLetter**(char x); bool **isKeyWord**(string str); bool **isBiOperator**(string str); bool **isMoOperator**(char x); bool **isDelimiter**(char x); bool **isState**(char x); bool **isFinalState**(char x); bool **isExist**(char x, NFAstateSet now);	词法分析识别各字符
void **NFABuild**(string grammarFileName);	构建 NFA
void **getExpandClosure**(NFAstateSet &now);	构建 NFA 的 ε -闭包
void **NFAConvert**();	NFA 转换为 DFA
void **scanFile**();	扫描源程序文件，进行词法分析
void **work**(string file_name);	词法分析工作函数
Void **outputParseResult**(string file_name);	词法分析结果

（2）语法分析：

重点函数和变量	功能
string _grammarFileName;	语法文法文件名
vector<_Production> _productions;	产生式集合
map<_Symbol, set<_Symbol>> _firstSet;	构建的 first 集合

_ActionGoTo _actionGoTo;	LR(1)文法构建的 action 和 goto 表
stack<_Symbol *> _symStack;	符号栈
stack<int> _staStack;	状态栈
vector<_Var> _varTable;	变量表
vector<_Fun> _funTable;	函数表
_MediateCodeGen _code;	中间代码
vector<pair<int, string>> fun_enter;	函数入口
_Symbol *_PopStack();	出栈函数
void _PushStack(_Symbol *_symbol);	入栈函数
void _ReadProductions();	读取文法产生式
void _GetFirstSet();	获得非终结符的 First 集
void _BuildDfa();	构建 DFA
void _SemanticAnalyse(list<Word> &_lexicalResult);	语义分析过程
_ProgramSet _GetClosure(_Program &_program);	获得闭包
bool _IsTs(*const* string &_symbol);	判断是否是终结符

（3）基本块划分

重点函数和变量	功能
vector<_Quaternary> inter_codes;	中间代码集合
map<string, vector<BaseBlock>> base_fun_blocks;	基本函数块集合
map<string, vector<InfoBlock>> info_fun_blocks;	带信息的函数块集合
map<string, vector<set<string>>> fun_inls;	函数块入口活跃变量集
map<string, vector<set<string>>> fun_outls;	函数块出口活跃变量集
vector<pair<int, string>> fun_enters;	函数入口集合
BlocksGenerator(vector<_Quaternary> &inter_codes, vector<pair<int, string>> &fun_enters);	基本块构造函数
void get_base_fun_blocks();	获得基本块
void get_info_fun_blocks();	获得信息基本块
string get_block_name();	获得基本块的名称

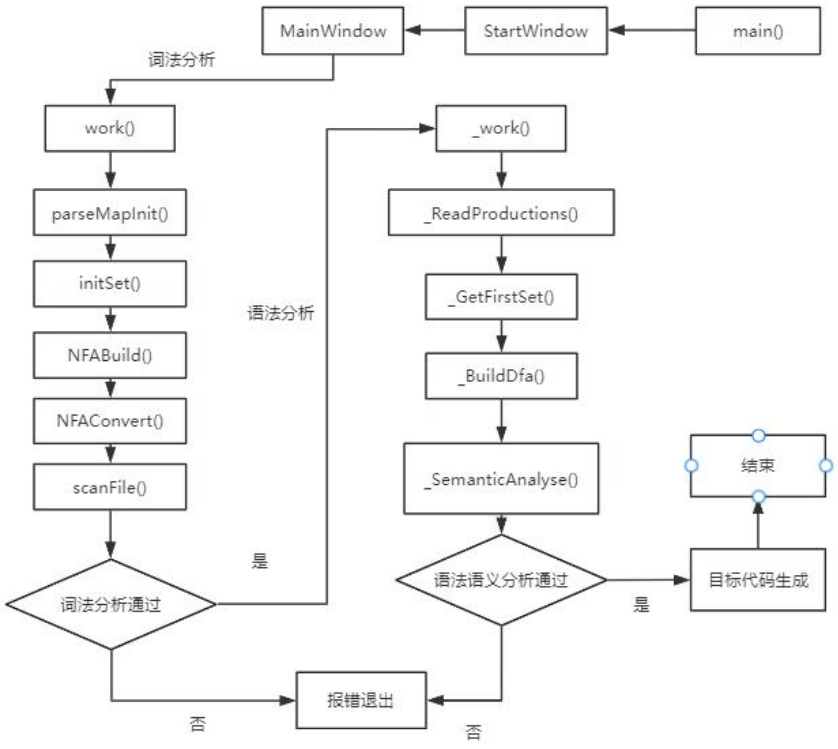
（4）Member 寄存器分配

重点函数和变量	功能
map<string, set<string>> R_Value;	R_Value 表
map<string, set<string>> A_Value;	A_Value 表
map<string, int> var_mem_pos;	变量存储在内存的位置表
int top_ptr;	top 栈帧
list<string> free_regs;	空闲寄存器表
MemManager();	寄存器管理函数
string **get_free_reg**(string &now_fun, map<string, vector<set<string>>> &fun_outls, map<string, vector<InfoBlock>> &info_fun_blocks, vector<InfoQuaternary>::iterator &cur_quaternary, vector<InfoBlock>::iterator &cur_info_block, vector<string> &object_codes);	获得一个空闲寄存器
string **get_reg_for_src**(string &now_fun, map<string, vector<set<string>>> &fun_outls, map<string, vector<InfoBlock>> &info_fun_blocks, vector<InfoQuaternary>::iterator &cur_quaternary, vector<InfoBlock>::iterator &cur_info_block, vector<string> &object_codes, string src);	这里需要传入变量是因为里面会有加载 src 到寄存器内的指令
string **get_reg_for_des**(string &now_fun, map<string, vector<set<string>>> &fun_outls, map<string, vector<InfoBlock>> &info_fun_blocks, vector<InfoQuaternary>::iterator &cur_quaternary, vector<InfoBlock>::iterator &cur_info_block, vector<string> &object_codes);	为目标变量获取空闲寄存器
void **store_var**(vector<string> &object_codes, string reg, string var);	把寄存器内的数据存入到内存中
void **store_outl_var**(vector<string> &object_codes, set<string> &outls);	存储出口活跃变量到内存中
void **clear_var_reg**(string var);	清空变量所在寄存器

（5）中间代码生成

重点函数和变量	功能
string now_fun;	当前函数名称
vector<InfoQuaternary>::iterator cur_quaternary;	当前四元式
vector<InfoBlock>::iterator cur_info_block;	当前基本块
vector<string> object_codes;	目标代码
ObjectCodeGenerator(vector<_Quaternary> &inter_codes, vector<pair<int, string>> &fun_enters); void **generate_code**();	一层循环
void **generate_code_for_fun**(map<string, vector<InfoBlock>>::iterator fun_iter);	二层循环：基于函数块
void **generate_code_for_block**(int block_idx);	三层循环：基于基本块
void **generate_code_for_quaternary**(int block_idx, int &var_nums, int &paras_nums, list<pair<string, bool>> &paras_list);	四层循环：基于四元式

2. 函数调用图



四、调试分析

1.测试数据：

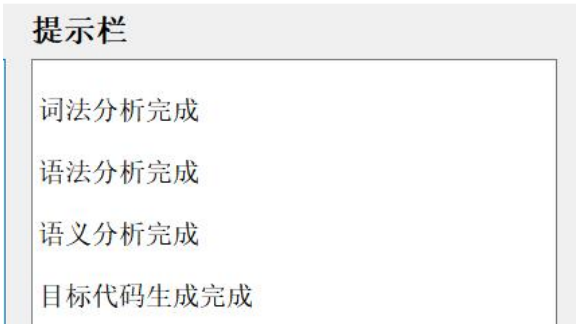
我们采用测试文件 0.txt，进行测试。

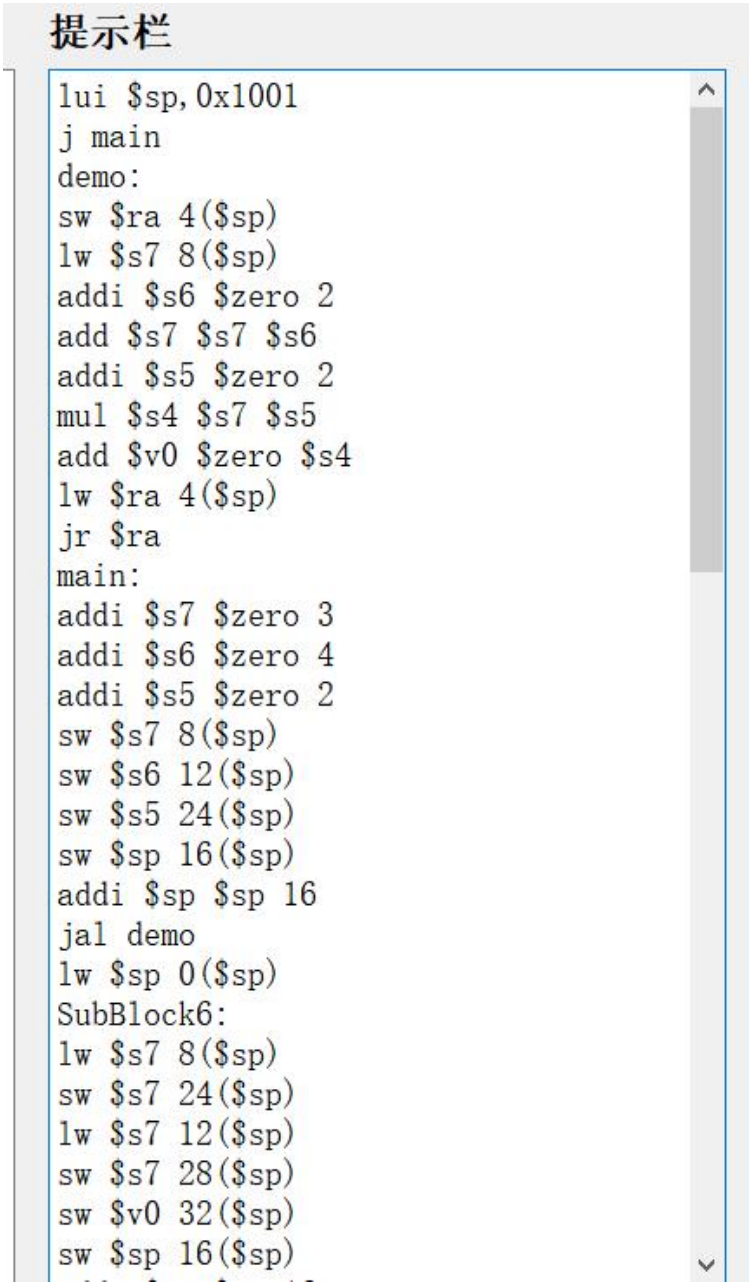
测试程序：

```
int a;
int b;
int program(int a,int b,int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
    {
        i=j*2;
        j=i;
    }
    return i;
}

int demo(int a)
{
    a=a+2;
    return a*2;
}

void main(void)
{
    int a;
    int b;
    int c;
    a=3;
    b=4;
    c=2;
    a=program(a,b,demo(c));
```

```
    return;
}
$
```

测试结果：

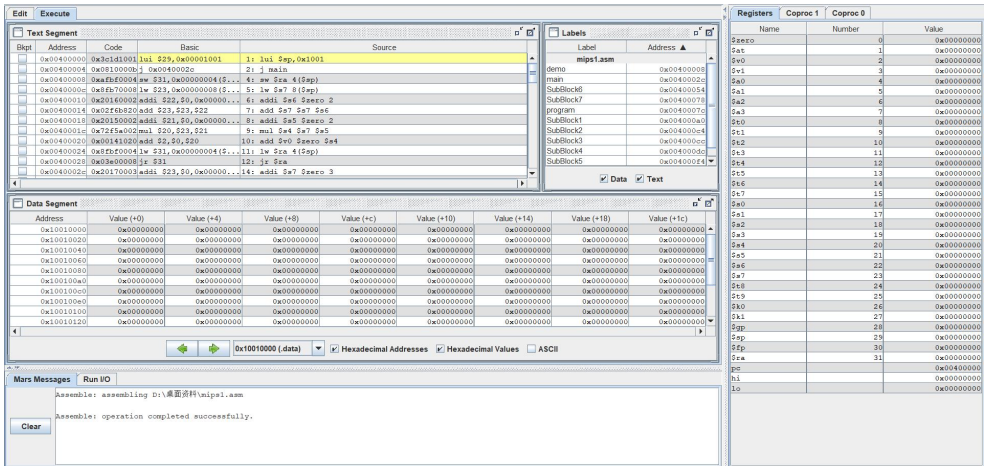

提示栏
词法分析完成
语法分析完成
语义分析完成
目标代码生成完成

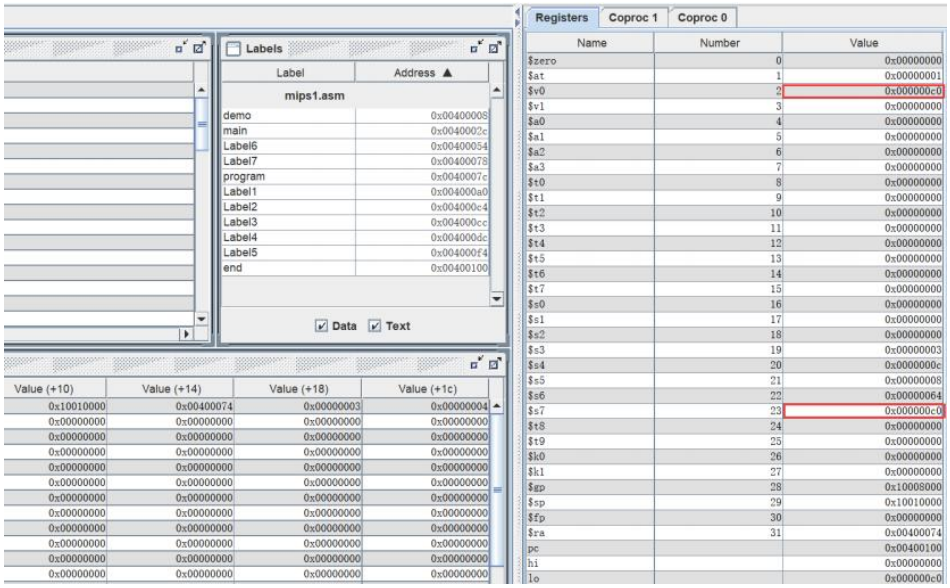

提示栏
lui $sp,0x1001
j main
demo:
sw $ra 4($sp)
lw $s7 8($sp)
addi $s6 $zero 2
add $s7 $s7 $s6
addi $s5 $zero 2
mul $s4 $s7 $s5
add $v0 $zero $s4
lw $ra 4($sp)
jr $ra
main:
addi $s7 $zero 3
addi $s6 $zero 4
addi $s5 $zero 2
sw $s7 8($sp)
sw $s6 12($sp)
sw $s5 24($sp)
sw $sp 16($sp)
addi $sp $sp 16
jal demo
lw $sp 0($sp)
SubBlock6:
lw $s7 8($sp)
sw $s7 24($sp)
lw $s7 12($sp)
sw $s7 28($sp)
sw $v0 32($sp)
sw $sp 16($sp)

汇编执行：

mips1.asm*

```
1  lui $sp,0x1001
2  j main
3  demo:
4  sw $ra 4($sp)
5  lw $s7 8($sp)
6  addi $s6 $zero 2
7  add $s7 $s7 $s6
8  addi $s5 $zero 2
9  mul $s4 $s7 $s5
10 add $v0 $zero $s4
11 lw $ra 4($sp)
12 jr $ra
13 main:
14 addi $s7 $zero 3
15 addi $s6 $zero 4
16 addi $s5 $zero 2
17 sw $s7 8($sp)
18 sw $s6 12($sp)
19 sw $s5 24($sp)
20 sw $sp 16($sp)
21 addi $sp $sp 16
22 jal demo
23 lw $sp 0($sp)
24 SubBlock6:
25 lw $s7 8($sp)
26 sw $s7 24($sp)
27 lw $s7 12($sp)
```





寄存器结果正确。

错误结果:

```
int max(int x,int y){
    if(x>y)
        return x;
    else
        return y;
}
int min(int x,int y){
    if(x<y){
        int t;
        t=x;
        x=y;
        y=t;
    }
    return y;
}
int sum(int x,int y){
    return x+y;
}
int main()
{
    int a;
    int b;
    int c;
    int d;
    int f;
    a=3;
    b=4;
    c=max(a,b);
    d=sum(a,b);
    f=min(c,d);
```

提示栏

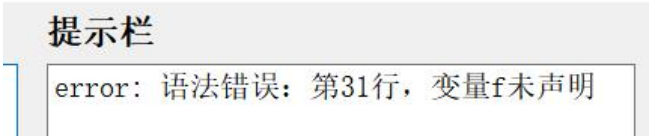
error: 分析失败，程序存在语法错误

```
int max(int x,int y){
    if(x>y){
        return x;
    }
    else{
        return y;
    }
}
int min(int x,int y){
    if(x<y){
        int t;
        t=x;
        x=y;
        y=t;
    }
    return y;
}
int sum(int x,int y){
    return x+y;
}
```

提示栏

error: 分析失败，程序存在语法错误

```
int max(int x,int y){
    if(x>y){
        return x;
    }
    else{
        return y;
    }
}
int min(int x,int y){
    if(x<y){
        int t;
        t=x;
        x=y;
        y=t;
    }
    return y;
}
int sum(int x,int y){
    return x+y;
}
int main()
{
    int a;
    int b;
    int c;
    int d;
    a=3;
    b=4;
    c=max(a,b);
    d=sum(a,b);
```



2.时间复杂度分析

（1）词法分析：

在词法分析阶段，NFA 转化成 DFA 的过程中，闭包操作是关键的算法步骤之一。当处理输入串长度为 k 时，闭包操作的时间复杂度取决于状态的出口数量（m）和状态的总数（n）。在闭包操作中，需要不断扩展当前状态以包含其可达的所有状态，这涉及到状态之间的转移和关联。因此，在整个转换过程中，闭包操作的复杂度为 O(k(m+n))，其中 k 表示输入串长度，m 表示状态的出口数量，n 表示状态的总数。

（2）语法分析：

在语法分析阶段，建立项目集的闭包操作是语法分析中的一个关键步骤。从

初始状态 I0 开始，考虑非终结符和终结符的数量分别为 m 和 n。在闭包操作中，需要考虑每个产生式的长度（k）以及产生式中的非终结符和终结符之间的关系。在最坏情况下，每个状态可能会生成 m+n-1 个新状态，这可能导致状态数量的快速增长。综合考虑产生式的复杂性和递归操作，建立项目集的闭包操作的时间复杂度为 O(n*m^2*k^2),其中 n 表示非终结符数量,m 表示终结符数量，k 表示产生式的长度。

（3）目标代码生成：

在目标代码生成阶段，处理活跃变量的时间复杂度较高，因为需要递归地查找活跃变量。活跃变量分析涉及从当前函数的基本块递归地向后查找，以确定哪些变量在后续代码中仍然活跃。每个递归调用需要考虑基本块的四元式，判断操作数是否活跃。在函数和基本块数量较多的情况下，获得活跃变量的复杂度为 O(m*2n)，其中 m 表示函数数量，n 表示基本块数量。特别是在基本块数量较大的情况下，处理活跃变量的复杂度会显著增加。

3.模块设计过程中的思考：

（1）词法分析：

在设计词法分析器时，是否使用单独构造识别标识符的 DFA 是一个权衡的问题。在某些情况下，可以直接在逐个字符扫描源程序，因此不必构造 DFA。然而，在本次设计中我们构造了 DFA。

通过构造 DFA，可以将标识符的识别过程与其他词法单元的处理过程进行统一。这样可以保持词法分析器的设计规范性和一致性，使得每个词法单元都有其独立的定义和识别规则。

构造 DFA 还可以将标识符的规则定义在文法中，并且可以相对容易地修改和扩展这些规则。将标识符的识别规则直接硬编码到扫描源程序的过程中，会导致修改或扩展规则变得复杂和繁琐。

将标识符的识别过程抽象为一个单独的 DFA，可以提高代码的可读性和可维护性。DFA 可以以图形化的方式表示，更容易理解和调试。将标识符的识别过程与其他词法单元的处理过程分开，可以降低代码的复杂性，使得代码更易于维护

和修改。

（2）语法分析：

在语法分析器的设计，我们考虑将文法中的每个符号定义为一个子类继承自符号类。

通过将每个符号定义为一个子类，可以在语法分析器中使用面向对象的编程思路，每个子类代表一个具体的符号，使得代码结构清晰明了。在每个子类中可以实现符号特定的行为和属性，从而使得语法分析器的逻辑更加清晰和易于理解。

这种方式也是一种使用面向对象的编程方式，可以充分利用继承、多态等特性，提高代码的可扩展性和可维护性。通过定义符号类的抽象方法和属性，可以方便地在子类中实现具体的语法规则和语义动作。

（3）目标代码生成：

在目标代码生成中，寄存器分配也是一个比较重要的问题，为了优化这一过程，可以使用高效的寄存器分配算法，如图染色算法或线性扫描算法等。这些算法可以在保证寄存器数量有限的情况下，将变量尽可能地分配到寄存器中，减少对内存的访问。

在寄存器分配过程中，还要考虑尽可能让寄存器能够复用。当一个变量的生命周期结束后，可以将其所占用的寄存器释放，以便给其他变量使用。

4. 模块调试过程：

（1）词法分析未能正确识别相应的词法单元。

```
void wordParse::scanFile()
{
    char info[100];
    char ch;
    int ptr;
    int keyFlag;
    ch = fgetc(fileSource);
    bool overFlag = false;
    while (!overFlag)
    {
        keyFlag = -1;
        ptr = 0;
        if (isdigit(ch)) // 多一个ch
        {
            keyFlag = 1;
            info[ptr++] = ch;
            ch = fgetc(fileSource);
            while (isLetter(ch) || isdigit(ch) || ch == '.' || ch == '+' || ch == '-')
            {
```

在设计过程中，发现有些词法单元的工作的时候未能正确识别，比如对于=和==的区别，<和<=之间的区别等等，都没有按照自己的预想进行正确的分析。自己检查了相应的判断函数，查询词法才找到问题。

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

（2）在 NFA 转化为 DFA 的过程出错。

在过程设计中，我们前面提到需要将 NFA 转化为 DFA。但是在最终的设计过程中却出现了错误。我们检查代码：

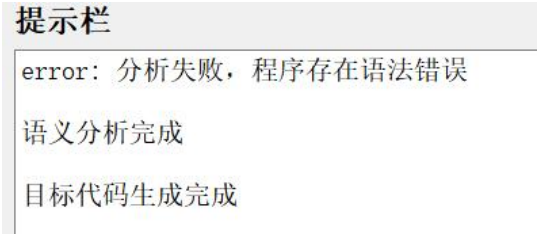
```
void wordParse::NFAConvert()
{
    DFAStateSetNums = 0;
    NFAstateSet nowStateSet;
    NFAstateSet nextStateSet;
    nowStateSet.states.insert(DFAStartState);
    stack<NFAstateSet> s;
    getExpandClosure(nowStateSet);
    s.push(nowStateSet);
    DFAStateSet[DFAStateSetNums++] = nowStateSet;
    memset(dfaTransfer, -1, sizeof dfaTransfer);
    for (int i = 0; i < 150; ++i)
        DFAIsFinal[i] = false;
    if (isFinalState(nowStateSet))
        DFAIsFinal[DFAStateSetNums - 1] = true;

    // 开始扩充
    while (!s.empty())
    {
        nowStateSet = s.top();
        s.pop();
        for (auto finalSignal : finalStateSet)
        {
            for (auto NTSignal : nowStateSet.states)
            {
                for (auto nextState : stateTransfer[NTSignal][finalSignal].states)    △array subscr
                {
```

经过检查之后发现在上述函数中并未处理ε转移的情况，在 NFA 中存在ε转移（空转移），转化为 DFA 时需要正确处理ε闭包操作。在上面的设计中出现了错误，仔细检查后才发现。

（3）出错之后，不会自动跳出，仍然会进一步分析。

在设计过程中，还出现了一种情况，就是已经报错之后，仍然未跳出程序，仍然继续执行，导致出现错误之后仍然继续执行导致程序异常退出。

比如最开始，我们的测试中有：



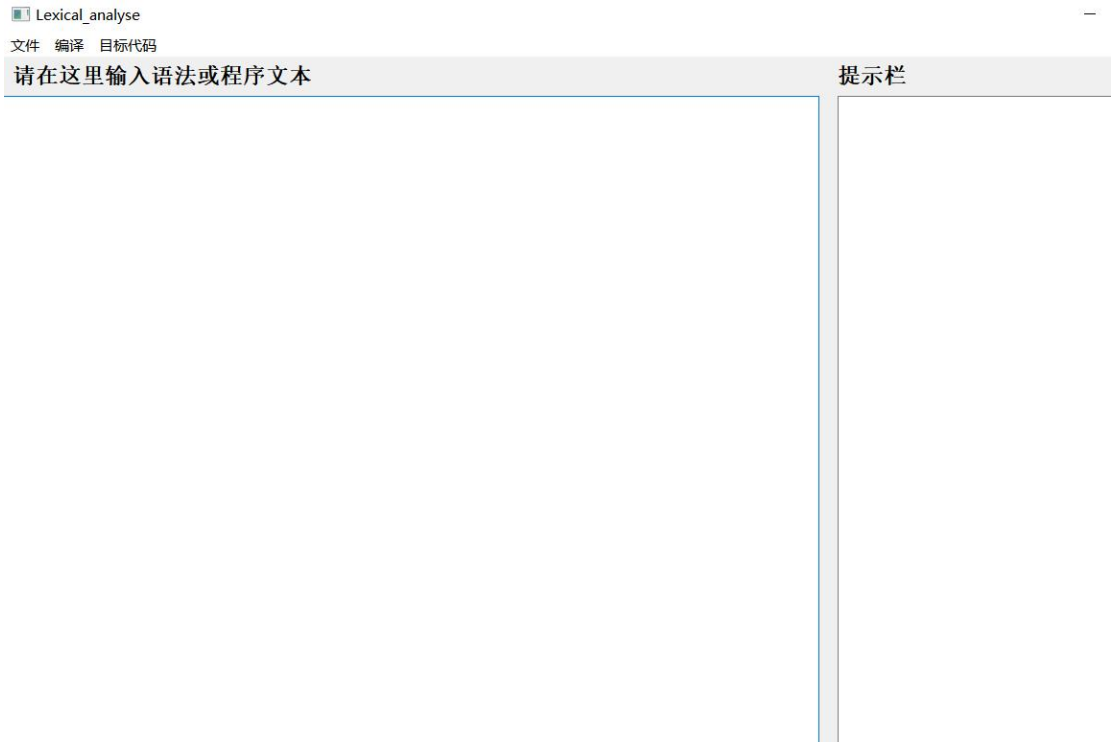
可以看到在出现错误之后，仍然继续执行了相关的代码，在有些测试样例中，就出现了程序异常的现象。

正确处理上述问题后，就可以了。

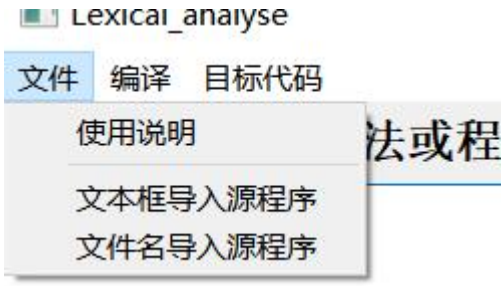
五、用户使用说明

1. 使用方法

点击 exe 文件之后，我们就可以使用了。进入画面为：



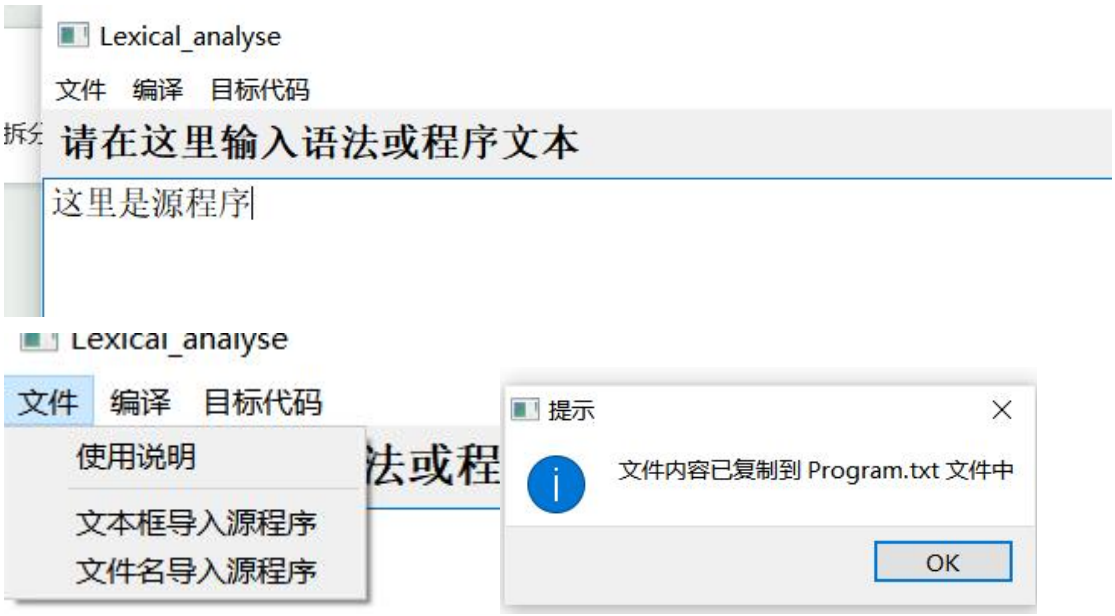
（1）使用说明：



点击使用说明之后，会在提示栏中显示帮助文档。

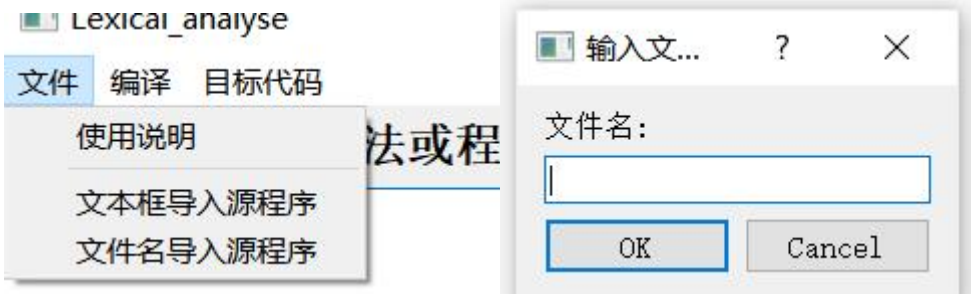
（2）文本框导入源程序：

在文本框中输入源程序后，点击文本框导入，出现提示，如下：



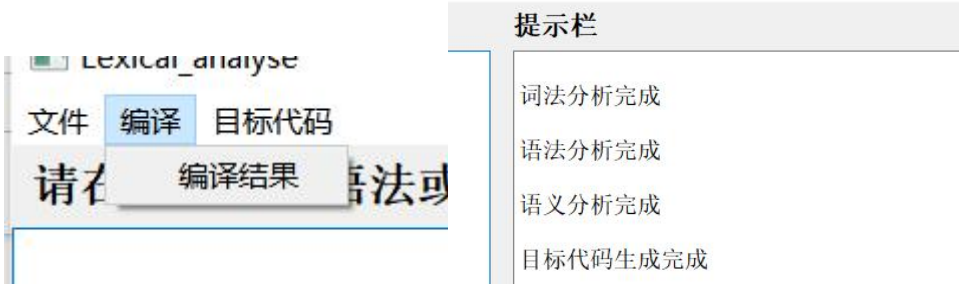
（3）文件名导入源程序：

点击文件源程序导入，出现提示，输入文件名后，就可以了，如下：



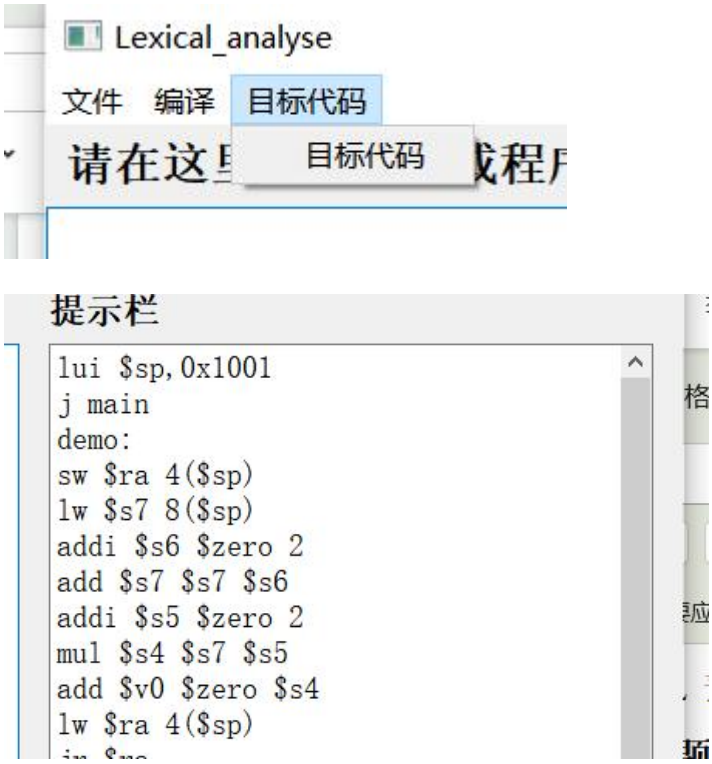
（4）编译：

导入源程序之后，就可以进行编译了，编译结果如下：



（5）目标代码：

编译源程序之后，就可以查看目标代码了，目标代码如下：



2. 实验环境

操作系统：Windows11 专业中文版

软件环境：Qt 5.14.2

编码语言：C++语言

六、实验总结

在经历了两个学期的学习和努力后，我成功地完成了编译原理课程设计的整个过程。这个过程中，我深刻认识到挑战和探索未知是个人成长和收获的关键。尽管我曾遇到困惑和挣扎，但正是这些艰难的时刻推动着我不断提升自己。学习应该是一个勇敢探索未知领域的旅程，而非止步于舒适区。特别是在计算机领域，技术的迅猛发展要求我们始终保持学习的状态，始终渴望获得新知，“活到老，学到老”，这样才能在不断变化的时代中保持竞争力。

在这次设计过程中，我意识到优秀的数据结构设计对于提高代码效率至关重要。通过与其他同学的交流和自己的探索，我学到了代码整洁规范和合理的数据结构设计的重要性。有些大佬的代码让我深刻认识到了 C++ STL 的强大之处。通过学习和应用这些容器，我加深了对 C++语言特性的理解，并提升了我的编程能力。

此外，我还发现代码实现与伪代码描述之间存在很大差异。掌握算法的流程固然重要，但更关键的是深入理解算法的原理，只有在细节处理上做到准确无误，才能实现正确的功能。在待用/活跃信息和寄存器分配策略设计中，由于对算法原理理解不够透彻，我犯了一些细节上的错误，这给我留下了深刻的教训。

整个实验过程不仅仅是对课堂知识的学习和巩固，更是一次宝贵的设计实践。我在语法分析的基础上完成了本次实验。虽然在编译过程中我并未进行过多的优化，只能识别部分 C 语言特性，但这次经历让我更深入地了解了编译原理，并让我相信设计和实现编译程序不再是遥不可及的任务。

总的来说，这次编译原理课程设计让我受益匪浅。不仅提升了我的技术能力，这次经历让我更加坚信，挑战困难、探索未知，才能不断成长和进步。这是一段宝贵的学习和成长之旅，让我更加自信地面对未来的挑战。我期待着未来的学习和发展，并相信我将在编译原理及其他领域取得更大的成就！祝愿我的学习和发展之路一切顺利！