

同濟大學

TONGJI UNIVERSITY

《编译原理》

课程设计实验报告

作业名称	类 C 编译器设计与实现
姓 名	吴达鹏 1751118
学院（系）	电子与信息工程学院
专 业	计算机科学与技术
任课教师	卫志华
日 期	2023 年 5 月 15 日

目录

一、 设计要求	3
二、 需求分析	3
2.1 程序的输入极其范围	3
2.2 输出形式	5
2.3 程序功能描述	7
2.4 测试数据	7
三、 概要设计	11
3.1 任务的分解	11
3.2 数据类型的定义	11
3.3 主程序流程	20
3.4 各模块工作流程	21
四、 详细设计	24
4.1 重点函数的重点变量	24
4.2 函数调用图	39
五、 调式分析	41
5.1 测试数据	41
5.2 时间复杂度分析	46
5.3 模块设计时的思考	47
5.4 调试时遇到的问题及解决方法	47
六、 用户使用说明	57
6.1 编译项目	57
6.1.1 cmake 打包生成 makefile	57
6.1.2 编译	58
6.2 程序运行	58
七、 总结与收获	60
7.1 总结	60
7.1.1 词法分析	60
7.1.2 语法语义分析	60
7.1.3 优化	60
7.1.4 目标代码生成	60
7.2 体会与收获	61

一、设计要求

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。

二、需求分析

2.1 程序的输入极其范围

1. 源程序。类 c 语言的源程序，除了一般的加减运算、bool 运算、赋值运算外还支持 while、if (else) 语句及过程调用，下面给出测试源程序 (后面的输出均来自于该源程序 TestProgram.txt)：

```
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;
}
$

```

2. 词法识别标识符的文法

```

101
S->_A
S->aA
S->bA
S->cA
.....
A->@
A->a
A->b
A->c
A->d
.....
A->_A
A->aA
A->bA
A->cA
A->dA

```

3. 语法的文法

```

S -> P
P -> N declare_block
value_expression -> sub_expression
value_expression -> sub_expression > sub_expression
value_expression -> sub_expression < sub_expression
value_expression -> sub_expression == sub_expression
value_expression -> sub_expression >= sub_expression
value_expression -> sub_expression <= sub_expression
value_expression -> sub_expression != sub_expression
sub_expression -> subsub_expression
sub_expression -> subsub_expression + sub_expression
sub_expression -> subsub_expression - sub_expression
.....

```

2.2 输出形式

1. 词法分析结果。若分析成功，输出为若干<值，类型>的键值对，其中，ID、NUM 分别表示标识符和数，其余的包括关键字、界符、运算符，采用一符一种的管理方式。

```

int INT
a ID
; SEMICO
int INT
b ID
; SEMICO
int INT
program ID
( LPAREN
int INT
a ID
, COMMA
int INT
b ID
.....
详见 parseResult.txt

```

1. 语法分析结果。若语法检查和语义检查通过，控制台直接输出中间代码（四元式），否则报错返回。

```

语法分析成功
( j , _ , _ , 20 )
( get , _ , _ , a )
( get , _ , _ , b )
( j<= , a , b , 6 )
( return , a , _ , _ )
( j , _ , _ , -1 )
( return , b , _ , _ )
( get , _ , _ , a )
( get , _ , _ , b )
( + , a , b , T0 )
( return , T0 , _ , _ )
( get , _ , _ , c )
( j<= , c , 1 , 18 )
( * , sum , c , T1 )
( = , T1 , _ , sum )
( - , c , 1 , T2 )
( = , T2 , _ , c )
( j , _ , _ , 12 )
( = , sum , _ , c )

```

```

( return , _ , _ , _ )
( = , 3 , _ , a )
( = , 4 , _ , b )
( para , a , _ , _ )
( para , b , _ , _ )
( call , add , _ , _ )
( = , RETURN , _ , T3 )
( para , a , _ , _ )
( para , T3 , _ , _ )
( call , max , _ , _ )
( = , RETURN , _ , T4 )
( = , T4 , _ , a )
( para , c , _ , _ )
( call , factorial , _ , _ )
( return , 0 , _ , _ )

```

2.3 程序功能描述

输入一个 c 语言源程序，对其进行词法分析、语法分析，并生成中间代码和 mips 目标代码。

2.4 测试数据

除面给出的一个正确测试程序外，再给出三个错误测试程序：
(错误 1 TestWrong.txt，if-else 语句块必须写在内)

```

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);
        return 0;
    }
    $

```

(错误 2TestWrong1.txt, 未声明的变量 f)

```

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);

    f=min(c,d);
    return 0;
}
$

```

(错误 3TestWrong3.txt, min 函数参数不匹配)

```

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(a,c,d);
    return 0;
}
$

```

三、概要设计

3.1 任务的分解

1. 词法分析

- (a) 初始化自定义的数据结构，包括关键字、运算符、界符等；
- (b) 跳过空格和换行，逐个字符读取源文件；
- (c) 根据字符读入的类型判断该词的类型，如果属于标识符，使用给定的文法构造的 DFA 判断其是否合法；
- (d) 输出 < 值, 类型 > 键值对序列。

2. 语法分析及语义分析（中间代码生成）

- (a) 读入给定的语法的文法，获得所有符号的 first 集
- (b) 从文法的第一条产生式的闭包开始扩展，获取 action 表和 goto 表，生成项目集规范族（DFA）。
- (c) 根据 DFA 和词法分析的输出结果，对源程序进行归约。与语法分析同时完成，归约的同时加上该类型语句语义动作（语法制导翻译）

3. 目标代码生成。根据中间代码生成可汇编执行的目标代码（mips 架构）。

- (a) 根据中间代码划分基本块
- (b) 获得各基本块的出口活跃变量和入口活跃变量
- (c) 根据变量是否活跃来生成寄存器分配策略
- (d) 结合 mips 汇编代码的基本知识，完成汇编代码的生成

3.2 数据类型的定义

1. 词法分析

```
typedef enum
{
    LCOMMENT, // 行注释
    SCOMMENT, // 段注释
    NEXTLINE, // 空白行
    ENDFILE,  // 文件结束
    ERROR,    // 错误码
    ID,
    NUM,
    // 关键字
    INT,
```

```

    VOID,
    RETURN,
    WHILE,
    ELSE,
    // 界符
    SEMICO,
    COMMA,
    .....
}WordType;
struct NFAsateSet
{
    multiset<char> states;
};
map<string, WordType> parseMap; // 字符串到词类型的映射, WordType
→ 为词类型的枚举
list<Word> parseResult;          // 所有 < 值, 属性 > 的键值对序列,
→ 其中 Word 定义为

                                   //pair<string, WordType>
set<string> keyWordsSet;          // 关键字集合
set<string> biOperatorSet;        // 运算符 (多符号运算符, 如
→ >=, !=, == 等) 集合
set<char> moOperatorSet;          // 运算符 (单符号运算符, 如 =, >, <
→ 等) 集合
set<char> delimiterSet;          // 界符结合
set<char> stateSet;              // DFA 状态集合
set<char> finalStateSet;         // DFA 终态集合
char DFASartState;              // 初始态
bool DFAIsFinal[STATE_NUMS];   // DFAIsFinal[i]=1 表示第 i 个状态
→ 是终态

/*NFA 的状态转移, stateTransfer[A][a] 表示 A 状态在接受 a 字符后可
→ 能达到的状态集合 */
NFAsateSet stateTransfer[STATE_NUMS][STATE_NUMS];
NFAsateSet DFASateSet[STATE_NUMS];    //存储由 NFA 转化成的 DFA
→ 的状态

```

```

int DFAStateSetNums = 0;           //DFA 状态数量
int dfaTransfer[STATE_NUMS][STATE_NUMS]; //DFA 的状态转移

```

2. 语法分析（只列出主要数据结构）

// 枚举操作类型

```
enum __Operator
```

```

{
    shift, // 移进
    reduct, // 规约
    acc, // 接受
    error // 报错
};

```

/* 文法中的一个符号为一个 *_Symbol* 对象 */

```
class _Symbol
```

```

{
public:
    bool _isTs; // 是否为终结符
    string _content; // 内容
    _Symbol();
    _Symbol(const string &_content, bool _isTs = false);
    friend bool operator<(const _Symbol &s1, const _Symbol &s2);
    friend bool operator==(const _Symbol &s1, const _Symbol &s2);
};

```

/* 一个 *_Production* 对象对应文法中的一条产生式 */

```
class _Production
```

```

{
public:
    _Symbol _left; // 产生式左部
    vector<_Symbol> _right; // 产生式右部
    int _idx; // 标号
};

```

/* 一个 *_Program* 对象为一个项目，即产生式 + 圆点 + 展望符 */

```

class _Program
{
public:
    int _pIdx;           // 对应的产生式标号
    int _dotPos;         // dot 所处的位置
    set<_Symbol> _outLook; // 展望符集合
    _Program();
    _Program(int _pIdx, int _dotPos, const set<_Symbol> &_outLook);
    friend bool operator<(const _Program &_s1, const _Program &_s2);
    friend bool operator==(const _Program &_s1, const _Program &_s2);
};

/* 项目集，一个 _ProgramSet 对象为多个项目的集合 */
class _ProgramSet
{
public:
    set<_Program> _program;
};

/* 一个动作，包括动作的类型和转移到的下一个状态 */
class _Action
{
public:
    __Operator _operator; // 操作
    int _nextState;       // 下一个状态
    _Action();
    _Action(__Operator _operator, int _nextState);
};

/* 项目集规范族，即识别给定文法的 DFA*/
class _Dfa
{
public:
    list<_ProgramSet> _stateSet; // 状态集
    map<_GoTo, int> _trans;      // 状态转移
};

```

```

};

class _GrammarParse
{
public:
    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

    /*typedef map<_GoTo, _Action> _ActionGoTo;*/
    _ActionGoTo _actionGoTo;           // LR1 文法对应的
    ↪ actionGoTo 表
    .....
}

```

3. 语义分析（中间代码生成）对于文法中的每一个符号，都定义为一个子类，继承 `_Symbol` 类，符号压栈时均为 `_Symbol` 类，弹栈时对其进行类型转换，过程较为清晰；如：

```

// ID 和常量类继承
class _ID : public _Symbol
{
public:
    string _name;
    _ID(const _Symbol &_symbol, const string &_name);
};

class _NUM : public _Symbol
{
public:
    string _val;
    _NUM(const _Symbol &_symbol, const string &_val);
};

class _Parameter : public _Symbol
{

```

```

public:
    list<__DataType> _parList;
    _Parameter(const _Symbol &_symbol);
};

class _SentenceBlock : public _Symbol
{
public:
    list<int> _nextList;
    _SentenceBlock(const _Symbol &_symbol);
};
.....

```

另外还有一些重要的数据结构和类的定义：

```

// 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
};

// 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
};

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
};
// 变量和函数
class _Var
{
public:
    string _varName;      // 变量名称
    __DataType _varType; // 变量类型
    int _varLevel;        // 变量层级
    friend bool operator==(const _Var &_v1, const _Var &_v2);
};

class _Fun
{
public:
    string _funName;      // 函数名称
    __DataType _returnType; // 返回值类型
    list<__DataType> _parType; // 所有参数的类型
    int _enterPoint;      // 函数入口
    friend bool operator==(const _Fun &_f1, const _Fun &_f2);
};

```

4. 目标代码生成

```

/* 基本块 */
struct Block
{
    string name;           //名称
    vector<_Quaternary> quaters; //四元式
    int next1;             //出口一
    int next2;             //出口二
};

```

```

/* 带有待用信息/活跃信息的基本块 */
struct ActiveInfo
{
    int next { -1 }; //待用四元式的编号，初
    ↪ 始为-1，表示非待用
    bool active { 0 }; //是否活跃，初始化为否
    ActiveInfo(){}
    ActiveInfo(int next, bool active){
        this->next = next;
        this->active = active;
    }
    ActiveInfo(const ActiveInfo& activeInfo){
        this->next = activeInfo.next;
        this->active = activeInfo.active;
    }
};

struct QuaternaryWithInfo
{
    _Quaternary q; //四元式代码
    ActiveInfo lop; //左操作数活跃信息
    ActiveInfo rop; //右操作数活跃信息
    ActiveInfo des; //目的操作数活跃信息
    QuaternaryWithInfo(){}
    QuaternaryWithInfo(_Quaternary q){
        this->q = q;
    }
    friend ostream& operator<<(ostream& out, QuaternaryWithInfo
    ↪ qWithInfo);
};

//带有四元式及变量活跃信息的基本块
struct BlockWithInfo
{
    string name;
    vector<QuaternaryWithInfo> quatersWithInfo;

```

```

    int next1;
    int next2;
    BlockWithInfo(Block block){
        this->name = block.name;
        this->next1 = block.next1;
        this->next2 = block.next2;
        quatersWithInfo.resize(block.quaters.size());
    }
};

class ObjectCodeGen
{
private:
    _Label label;
    //基本块名称
    vector<string> objectCodes;
    //目标代码
    map<string, vector<Block>> funcBlocks;
    //函数基本块
    map<string, vector<BlockWithInfo>> funcBlocksWithInfo;
    //带有待用/活跃信息的函数基本块
    map<string, list<ActiveInfo>> activeInfos;
    //待用/活跃信息表, 用 {函数名 + 变量名, 活跃信息列表} 作为映射
    map<string, vector<set<string>>> outStillActive;
    //各函数出了基本块仍然活跃的变量
    set<string> inActive;
    //当前块的入口活跃变量
    map<string, int> offsets;
    //变量偏移位置, 用于生成汇编代码变量在内存中存储时的偏移量
    map<string, set<string>> RVALUE;
    //某个寄存器存放了哪些变量
    map<string, set<string>> AVALUE;
    //某个变量存放的位置 (寄存器名)
    string curFuncName;
    //当前分析的函数

```

```

vector<BlockWithInfo>::iterator curIBlock;
//当前分析的基本块
vector<QuaternaryWithInfo>::iterator curQuater;
//当前分析的四元式
list<string> freeRegs;
//空闲寄存器
int top;
//栈顶指针
map<string,bool> flag;
//递归控制标识
.....
};

```

3.3 主程序流程

```

#include "wordParse.h"
#include "grammarParse.h"
#include "ObjectCodeGen.h"
int main()
{
    // 先调词法工作
    wordParse wp;
    wp.work();
    wp.outputParseResult("LexicalResult.txt");
    // 再调语法分析
    _GrammarParse _gp("grammar.txt");
    _gp._Work();
    //再调语义分析
    _gp._SemanticAnalyse(wp.getWordParseResult());

    //目标代码生成
    ObjectCodeGen objectCodeGen;
    objectCodeGen.divideBlock(_gp.getFunTable(),_gp.getMediateCode());
    objectCodeGen.handleBlock();
    objectCodeGen.gen();
    system("pause");
}

```

```

return 0;
}

```

3.4 各模块工作流程

整个程序分为 main、词法分析、语法语义分析、目标代码生成四个模块，各模块间通过输出文件关联，即词法分析输出的键值对序列作为语法分析的输入，语法分析结束后紧接着进行语义分析（语法分析、语义分析在同一模块内），然后将生成的中间代码交给目标代码生成器，各模块内部的工作流程如下：

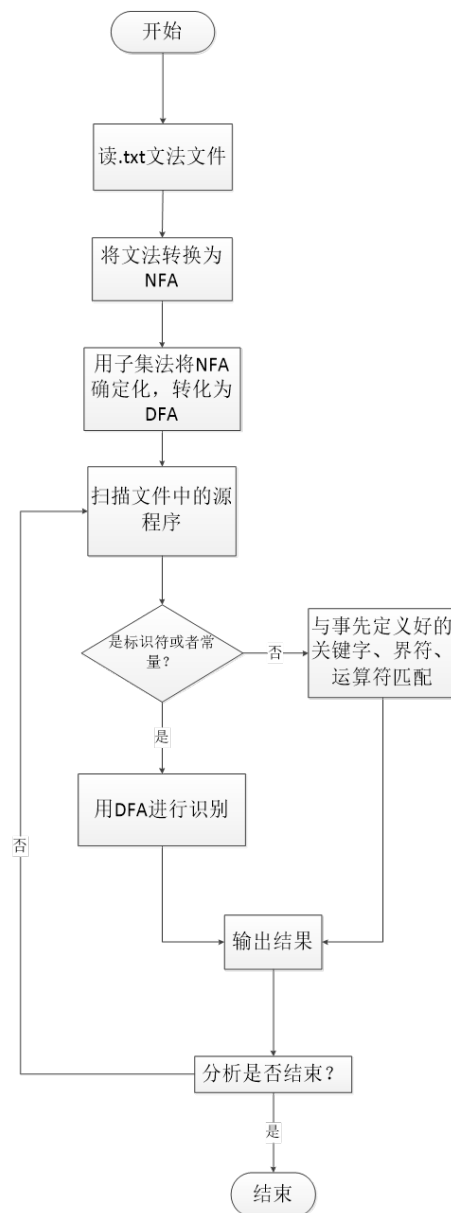


图1 词法分析工作流程

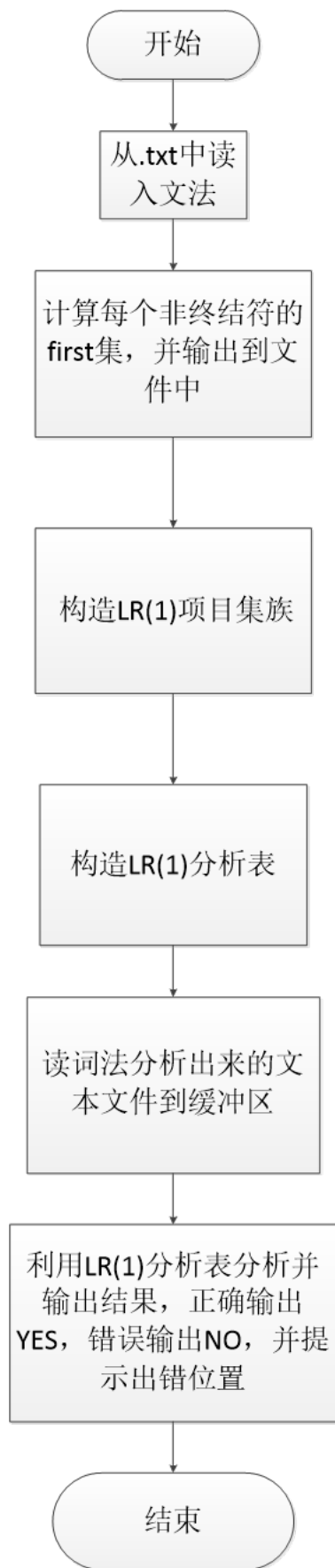


图2 语法分析工作流程

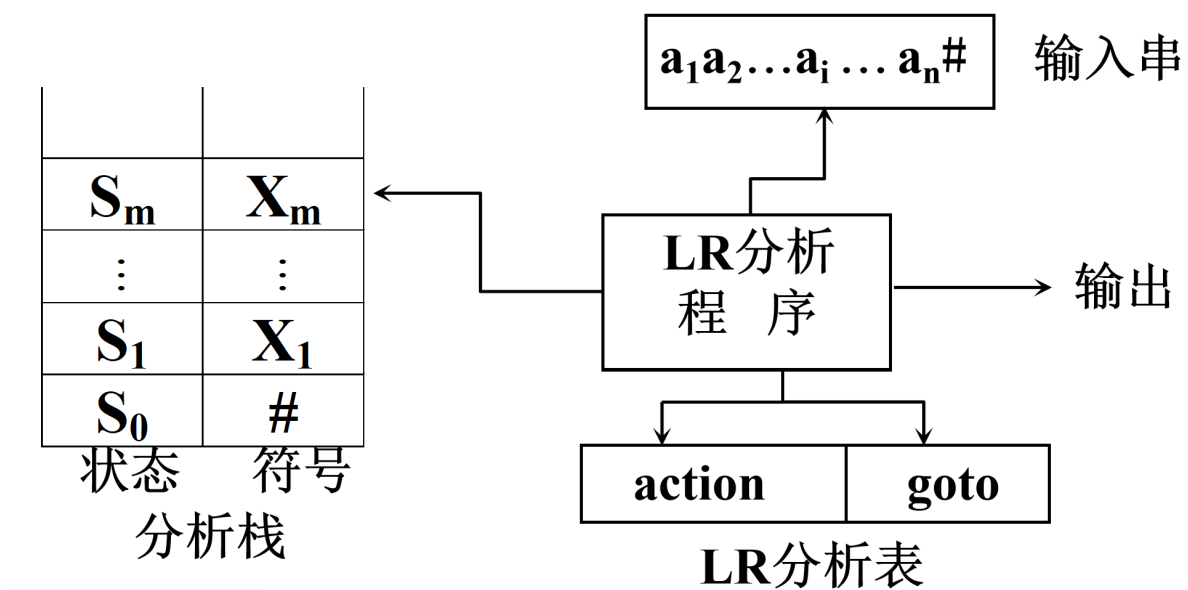


图3 语法（语义）分析工作原理

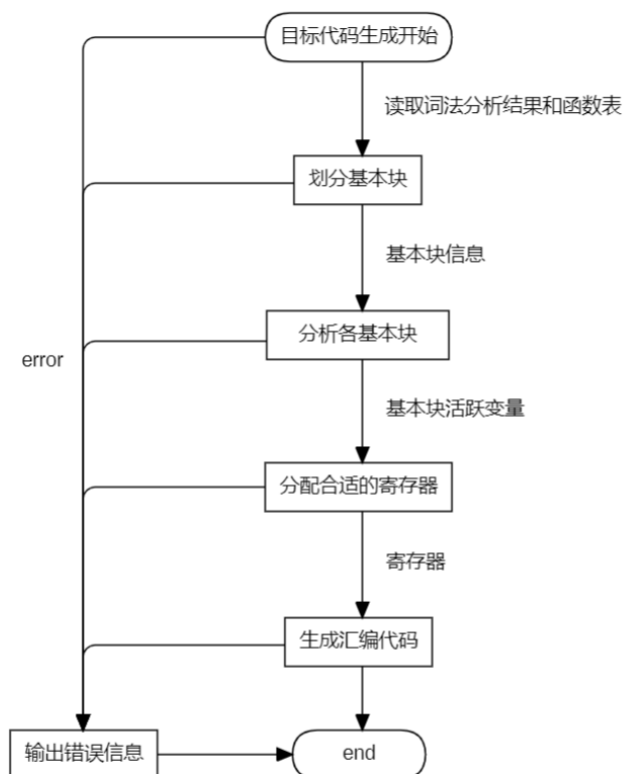


图4 目标代码生成工作流程

四、详细设计

4.1 重点函数的重点变量

1. 词法分析

- (a) void wordParse::scanFile()。扫描源程序，进行词法分析，输出 < 值, 属性 > 键值对序列。

```
void wordParse::scanFile()
{
    .....
    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 == '.')
            {
                keyFlag = 1;
                info[ptr++] = ch;
                ch = fgetc(fileSource);
            }
            info[ptr] = '\\0';
        }
        if (keyFlag == 1)
        {
            if (judgeConst(info))
            {
                // NUM
                parseResult.push_back({info, NUM});
            }
        }
    }
}
```



```

        // outputFile << 3;
    }
    else
    {
        cout << info << " "
             << " 出错, 不是常量" << endl;
    }
    ptr = 0;
    keyFlag = -1;
}
if (isLetter(ch))
{
    //识别标识符
    .....
}
if (keyFlag == 2)
{
    .....
}
if (isDelimiter(ch))
{
    //识别界符
    .....
}
if (isMoOperator(ch))
{
    //识别运算符
    .....
}
if (ch == '\r')
{
    .....
}
if (ch == ' ' || ch == '\n' || ch == '\t')
{

```

```

        .....
    }
}
}

```

2. 语法分析

- (a) `void _GrammarParse::_ReadProductions()`。读取给定的文法，将每条产生式保存在 `_productions` 容器中。

```

void _GrammarParse::_ReadProductions()
{
    // 读入产生式，记下标号
    int _idx = 0; // 从 0 开始
    // 打开文件
    ifstream _fin;
    _fin.open(_grammarFileName);
    if (!_fin.is_open())
    {
        _errorLog(" 文件打开失败");
        exit(-1);
    }
    // 读入产生式，这里要注意空行的处理
    char _readBuf[_READBUFLLEN];
    while (_fin >> _readBuf)
    {
        //每个 while 循环读一条产生式
        _Production _tmp;
        _tmp._left = _Symbol{string(_readBuf)};
        _tmp._idx = _idx++;
        _fin >> _readBuf;
        //将产生式中的 “->” 读掉，不需要保存
        _fin.getline(_readBuf, _READBUFLLEN);
        // 这里利用 stringstream 非常方便
        stringstream _tmpStream(_readBuf);
        while (_tmpStream >> _readBuf)
        {

```

```

        _tmp._right.push_back(_Symbol{_readBuf,
↪ _IsTs(_readBuf)}));
    }
    _productions.push_back(_tmp); // 压入
}
}

```

- (b) `void _GrammarParse::_GetFirstSet()`。获取每个非终结符的 *first* 集, 存放在 `_firstSet` 变量中, `_firstSet` 定义为 `map<Symbol,set<Symbol>>` 的映射, `_firstSet[A]` 表示符号 *A* 的 *first* 集。

```

void _GrammarParse::_GetFirstSet()
{
    // 对于每个非终结符求其 first 集

    // 设置一个终止变量, 默认值为 0, 如果 first 集有新的变动就继续循
↪ 环
    bool _finishFlag = 0;
    while (!_finishFlag)
    {
        _finishFlag = 1;
        // 遍历所有的产生式
        vector<_Production>::iterator _productionIter;
        for (_productionIter = _productions.begin();
↪ _productionIter != _productions.end(); _productionIter++)
        {
            // 获得当前产生式 ID, 左边和右边符号
            auto _nowIdx = _productionIter->_idx;
            auto _leftSymbol = _productionIter->_left;
            auto _rightSymbol = _productionIter->_right;

            // 考虑遍历产生式右部
            vector<_Symbol>::iterator _rightSymbolIter;
            for (_rightSymbolIter = _rightSymbol.begin();
↪ _rightSymbolIter
                != _rightSymbol.end(); _rightSymbolIter++)
            {

```



```

int _stateIdx = 0; // 状态标号
list<ProgramSet>::iterator _stateIter;
Program _firstPro;
_firstPro._dotPos = 0;
_firstPro._outLook.insert({"$", _IsTs("$)}));
_firstPro._pIdx = 0;
_dfa._stateSet.push_back({_GetClosure(_firstPro)});
// 将第一个产生式的闭包作为第一个状态

// 开始遍历状态
for (_stateIter = _dfa._stateSet.begin(); _stateIter
!= _dfa._stateSet.end(); _stateIter++, _stateIdx++)
{
    // 遍历这个状态集中的所有项目
    auto _nowProgramSet = _stateIter->_program;
    set<Program>::iterator _programIter;
    for (_programIter = _nowProgramSet.begin(); _programIter
!= _nowProgramSet.end(); _programIter++)
    {
        // 判断当前是否为归约项目
        auto _nowDotPos = _programIter->_dotPos;
        auto _nowProduction =
→ _productions[_programIter->_pIdx];
        if (_nowDotPos == _nowProduction._right.size())
        {
            // 归约项目，那么需要归约所有的 outlook
            auto _nowOutLook = _programIter->_outLook;
            for (auto &_s : _nowOutLook)
            {
                // 这些符号可以进行归约
                _Action _action({_reduct, _programIter->_pIdx});
                if (_programIter->_pIdx == 0)
                {
                    _action._operator = acc;
                }
            }
        }
    }
}

```



```

    for (_scanIter = _lexicalResult.begin(); _scanIter !=
↪ _lexicalResult.end();)
    {
        // 获得当前词的值和类型
        auto _nowSymbol = _scanIter->first;
        auto _nowType = _scanIter->second;
        if (_nowType == LCOMMENT || _nowType == SCOMMENT)
        {
            //跳过注释
            scanIter++;
            continue;
        }
        if (_nowType == NEXTLINE)
        {
            _scanIter++;
            _linesCnt++;
            continue;
        }
        _Symbol *_curSymbol;
        if (_nowType == ID)
        {
            _curSymbol = new _ID(_Symbol{"ID", true}, _nowSymbol);
        }
        else if (_nowType == NUM)
        {
            _curSymbol = new _NUM(_Symbol{"NUM", true}, _nowSymbol);
        }
        else
        {
            _curSymbol = new _Symbol(_nowSymbol, true);
        }

        // 下面就是根据符号来进行移进或者归约

        auto _nowSta = _staStack.top();

```

```

// 查表获得下一步
_GoTo _goTo(_nowSta, *_curSymbol);
if (_actionGoTo.count(_goTo) == 0)
{
    _errorLog(" 归约失败，找不到对应的 action");
    exit(-1);
}

auto _nowAction = _actionGoTo[_goTo];
if (_nowAction._operator == shift)
{
    // 移进
    _symStack.push(_curSymbol);
    _staStack.push(_nowAction._nextState);
    _scanIter++;
}
else if (_nowAction._operator == acc)
{
    _acc = true;
    break;
}
else
{
    //归约动作，归约的同时判断语义动作
    //=====//
    // 下面要实现语义分析，那么就需要对所有产生式进行分别操作
    auto _nowProIdx = _nowAction._nextState; // 获得当前产生式
    的 ID
    auto _nowPro = _productions[_nowAction._nextState];
    auto _popTimes = _nowPro._right.size();
    switch (_nowProIdx)
    {
        //共有 59 条产生式，对应 59 个 case
        case 1:
        {
            // P -> N declare_block

```



```

        // 这里首先需要弹出 N 和 declare_block
        // 然后将 main 的入口回填到 N
        _Fun *_f = _LookUpFunc("main");
        _PopStack();
        _N *_n = (_N *)_PopStack();
        _code._BackPatch(_n->_nextList, _f->_enterPoint);
        printf("enterpointer = %s", _f->_enterPoint);
        _Symbol *_p = new _Symbol(_nowPro._left);
        _PushStack(_p);
        break;
    }

    .....
}

}

```

4. 目标代码生成

- (a) `divideBlock()`，划分基本块。按照三个原则划分基本块：1、程序的第一个语句；
2、能由条件转移或无条件转移转移到的语句；3、紧跟在条件转移后面的语句。

```

void ObjectCodeGen::divideBlock()
{
    for(funcEnterTable->begin()~end())
    {
        /* 遍历每一个函数，以函数为单位，划分每一个函数的基本块 */

        //第一条语句，即函数的入口点，算第一个块
        blockEnters.push(curFunEntryPoint);

        //遍历当前函数的所有四元式
        for(int i = curFunEntryPoint; i != nextFunEntryPoint; i++)
        {
            _Quaternary q = interMediateCode[i];
            if(q._opt[0] == 'j')
            {

```

```

        //条件跳转语句，需要额外把紧跟其后的第一条语句（如果
→ 有的话）作为下一个块的入口点
        if(q._opt != "j" && i + 1 < nextFunEntryPoint)
            blockEnters.push(i + 1);

        //条件转移或无条件转移语句跳转到的语句
        blockEnters.push(stoi(q._rd));
    }
    else if((q._opt == "return" || q._opt == "call") && i +
→ 1 < nextFunEntryPoint)
    {
        //函数调用或是返回语句
        blockEnters.push(i + 1);
    }
}

//得到每个基本块的起始入口后，开始设置每一个块的属性
//包括名称、所包含的四元式以及它的两个出口（如果有的话）
.....
}

```

- (b) `handleBlock()`，处理每一个基本块，获得在该基本块之后活跃的变量信息，以便后面为其分配寄存器。

```

void ObjectCodeGen::handleBlock(){

    //1、初始化待用/活跃信息表，将该块后面出现的变量的活跃信息置为
→ true
    //从当前块向后面的逻辑块找
    for(auto iter = funcBlocks)
    {
        //所有出现过的变量都置为非待用
        //是否活跃需要根据后面的基本块是否出现在左边决定
    }
}

```

```
//这里保存基本块出口后仍然活跃的变量信息
```

```
outStillActive[iter->first].push_back(activeVar);
```

```
}
```

- (c) `gen()`, 生成四元式代码。更细化地, 遍历每一个函数块及函数块下的每一条四元式, 逐一生成四元式代码。其中, 每个函数块下的基本块处理时, 需要获取该基本块的入口活跃变量 (可以由前面求的出口活跃变量求来), 将其值存入 `AVALUE`, 否则将会报出变量引用前未初始化的错误。

```
void ObjectCodeGen::gen()
```

```
{
```

```
    objectCodes.push_back("lui $sp,0x1001");
```

```
    objectCodes.push_back("j main");
```

```
    for(auto iter = funcBlocksWithInfo.begin())
```

```
    {
```

```
        offsets.clear();
```

```
        curFuncName = iter->first;
```

```
        vector<BlockWithInfo>& blocks = iter->second;
```

```
        for(auto biter = blocks.begin()){
```

```
            curIBlock = biter;
```

```
            getInActiveVar(biter - blocks.begin(), blocks);
```

```
            for(auto citer = inActive.begin())
```

```
                AVALUE[*citer].insert(*citer);
```

```
            genForFuncBlock(curIBlock - blocks.begin());
```

```
            AVALUE.clear();
```

```
            RVALUE.clear();
```

```
        }
```

```

    }
    objectCodes.push_back("end:");

    outputObjectCode("ObjectCode.asm");
    cout << "Compile successfully! Please open the output file to
↪ check the result! " << endl;
}

```

- (d) `genForQuaternary()`, 为每条四元式生成目标代码。主要分为几种情况处理：1、跳转语句和条件跳转语句；2、参数传递语句处理；3、函数转子语句的处理；4、函数返回语句的处理；5、函数跳转时参数接收的处理；6、赋值语句的处理；7、计算语句的处理。

```

void genForQuaternary(){
.....
    if(op == "j")
        objectCodes.push_back(op + " " + curQuater->q._rd);
    else if(op[0] == 'j')
    {
        //条件跳转
        if(op == "j>=")
            op = "bge";
        else if(op == "j<=")
            op = "ble";
        else if(op == "j>")
            op = "bgt";
        else if(op == "j<")
            op = "blt";
        else if(op == "j==")
            op = "beq";
        else if(op == "j!=")
            op = "bne";
        string R1 = getReg(rs);
        string R2 = getReg(rt);
        objectCodes.push_back...
    }
    else if(op == "para")

```

```

{
    //传递参数
    par_list.push_back...
}
else if(op == "call")
{
    //转调函数
    for(auto iter = par_list.begin())
    {
        //分配寄存器, 将该参数压栈
        ...
    }

    arg_num = par_num = 0;
    par_list.clear();

    //更新 sp

    //跳转到对应函数

    //返回现场
}
else if(op == "return")
{
    //函数返回
    if(isdigit(rs[0])){
        objectCodes.push_back("addi $v0 $zero " + rs);
    }
    else if(isVar(rs)){
        auto aiter = AVALUE[rs].begin();
        if((*aiter)[0] == '$'){
            //如果是寄存器
            objectCodes.push_back("add $v0 $zero " + (*aiter));
        }
        else{

```

```

        //如果是在内存中
        objectCodes.push_back("lw $v0 " +
→ to_string(offsets[*aiter]) + "($sp)");
    }
}

if(curFuncName == "main"){
    //直接结束
    objectCodes.push_back("j end");
}

else{
    //返回调用处
    objectCodes.push_back("lw $ra 4($sp)");
    objectCodes.push_back("jr $ra");
}

}

else if(op == "get"){
    //函数传参 (子函数接收)
    offsets[rd] = top;
    top += 4;
    AVALUE[rd].insert(rd);
}

else if(op == "="){
    //赋值运算 A = B (=, @RETURN, _, _) (=, a, _, T1);
    string pos;
    if(rs == "@RETURN"){
        pos = "$v0";
    }
    else{
        pos = getReg(rs);
    }
    AVALUE[rd].erase(rd);
    //赋值过，就擦掉，如果是活跃变量，就要生成存数指令写入内存
    AVALUE[rd].insert(pos);
    RVALUE[pos].insert(rd);
    if(!curQuater->lop.active)

```

```

        releaseReg(rs);
    if(!curQuater->rop.active)
        releaseReg(rt);
}
else{
    // + - * /
    string R1 = getReg(rs);
    string R2 = getReg(rt);
    string Rd = getRegForRd();
    if(op == "+")
        objectCodes.push_back...
    else if(op == "-")
        objectCodes.push_back...
    else if(op == "*")
        objectCodes.push_back...
    else if(op == "/"){
        objectCodes.push_back...
    }

    AVALUE[rd].insert(rd);
    if(!curQuater->lop.active)
        releaseReg(rs);
    if(!curQuater->rop.active)
        releaseReg(rt);

}
}

```

4.2 函数调用图

本程序大体分为 4 个功能模块，主模块 main、词法分析模块 myPaser、语法语义分析模块 grammarPaser，objectCodeGen，由主模块分别调用词法分析器、语法语义分析器、目标代码生成器功能，词法分析器输出的键值对序列作为语法分析的输入，通过后与函数符号表再作为目标代码生成器的输入，各模块内部函数调用关系如下：

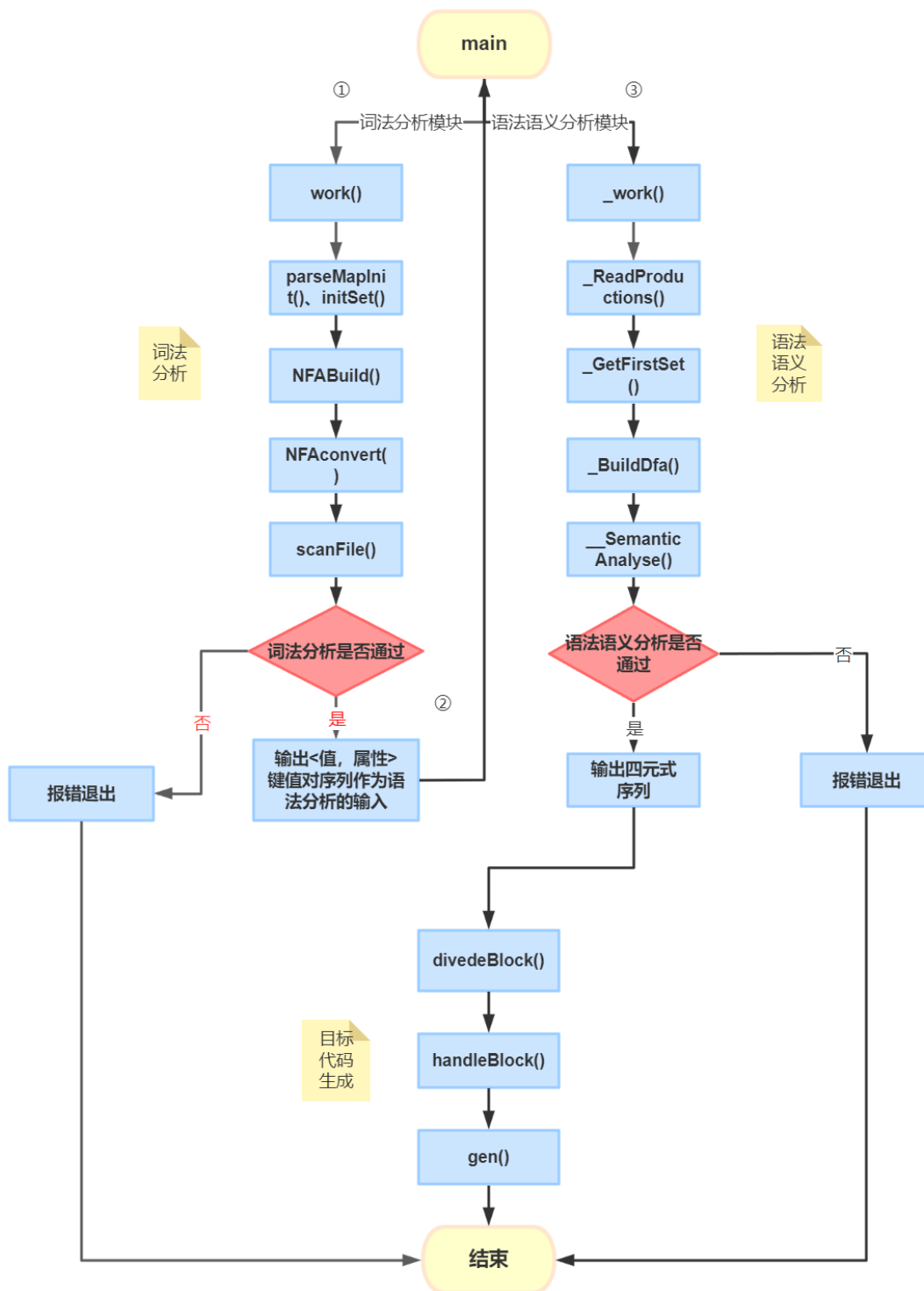


图 5 函数调用图

五、调式分析

5.1 测试数据

对于正确的测试程序，直接输出其四元式序列，对于错误的测试程序，报出相应的错误。例如，对于正确的确实程序 test.txt，输出其中间代码（四元式序列）和目标代码：

```
0 (j,_,_,24)
1 (get,_,_,a)
2 (get,_,_,b)
3 (get,_,_,c)
4 (=,0,_,i)
5 (+,b,c,T0)
6 (j<=,a,T0,12)
7 (*,b,c,T1)
8 (+,T1,1,T2)
9 (+,a,T2,T3)
10 (=,T3,_,j)
11 (j,_,_,13)
12 (=,a,_,j)
13 (j>,i,100,18)
14 (*,j,2,T4)
15 (=,T4,_,i)
16 (=,i,_,j)
17 (j,_,_,13)
18 (return,i,_,_)
19 (get,_,_,a)
20 (+,a,2,T5)
21 (=,T5,_,a)
22 (*,a,2,T6)
23 (return,T6,_,_)
24 (=,3,_,a)
25 (=,4,_,b)
26 (=,2,_,c)
27 (para,c,_,_)
28 (call,demo,_,_)
29 (=,@RETURN,_,T7)
30 (para,a,_,_)
31 (para,b,_,_)
32 (para,T7,_,_)
33 (call,program,_,_)
34 (=,@RETURN,_,T8)
35 (=,T8,_,a)
36 (return,_,_,_)
```

```

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 $s7 $s7 $s5
add $v0 $zero $s7
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)
Label6:
lw $s7 8($sp)
sw $s7 24($sp)
lw $s7 12($sp)
sw $s7 28($sp)
sw $v0 32($sp)
sw $sp 16($sp)
addi $sp $sp 16
jal program
lw $sp 0($sp)
Label7:
j end
program:
sw $ra 4($sp)
addi $s7 $zero 0
lw $s6 12($sp)
lw $s5 16($sp)
add $s4 $s6 $s5
sw $s7 20($sp)
lw $s3 8($sp)
ble $s3 $s4 Label2
Label11:
lw $s7 12($sp)

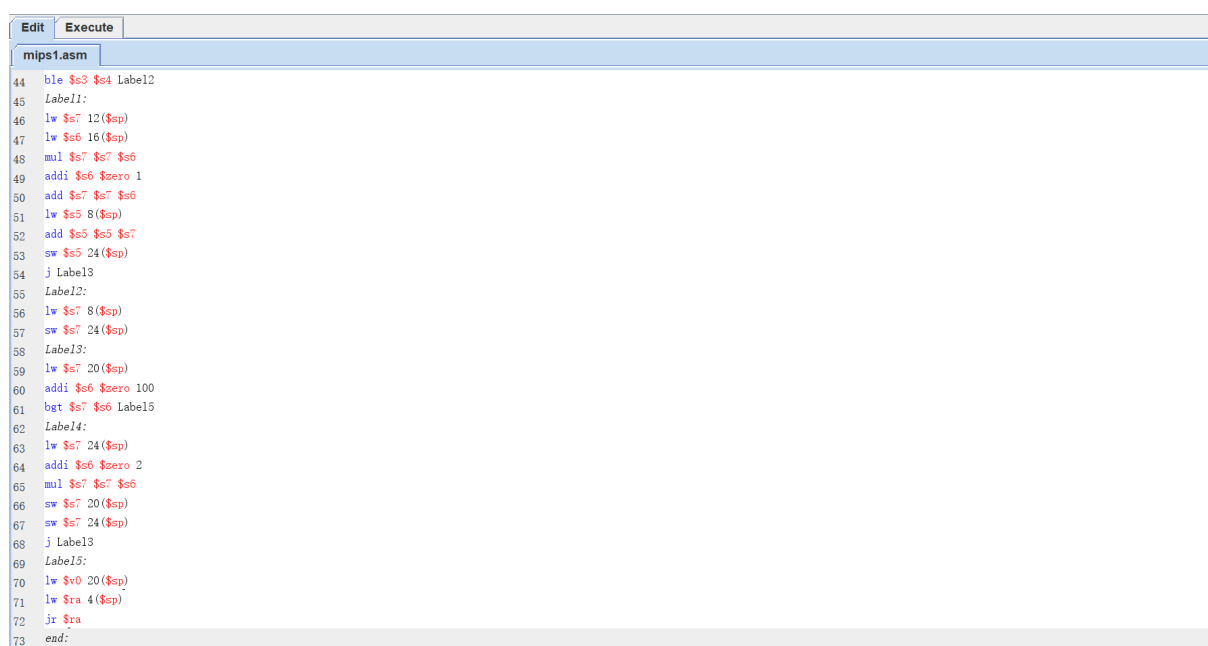
```

```

lw $s6 16($sp)
mul $s7 $s7 $s6
addi $s6 $zero 1
add $s7 $s7 $s6
lw $s5 8($sp)
add $s5 $s5 $s7
sw $s5 24($sp)
j Label3
Label2:
lw $s7 8($sp)
sw $s7 24($sp)
Label3:
lw $s7 20($sp)
addi $s6 $zero 100
bgt $s7 $s6 Label5
Label4:
lw $s7 24($sp)
addi $s6 $zero 2
mul $s7 $s7 $s6
sw $s7 20($sp)
sw $s7 24($sp)
j Label3
Label5:
lw $v0 20($sp)
lw $ra 4($sp)
jr $ra
end:

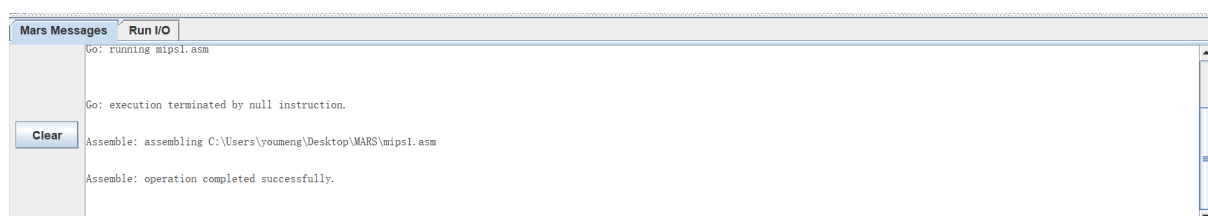
```

执行该汇编代码，得到 $v0,s7$ 寄存器的值为 $0xc0$ ，十进制为 192，答案正确。



```
44 ble $s3 $s4 Label2
45 Label1:
46 lw $s7 12($sp)
47 lw $s6 16($sp)
48 mul $s7 $s7 $s6
49 addi $s6 $zero 1
50 add $s7 $s7 $s6
51 lw $s5 8($sp)
52 add $s5 $s5 $s7
53 sw $s5 24($sp)
54 j Label3
55 Label2:
56 lw $s7 8($sp)
57 sw $s7 24($sp)
58 Label3:
59 lw $s7 20($sp)
60 addi $s6 $zero 100
61 bgt $s7 $s6 Label5
62 Label4:
63 lw $s7 24($sp)
64 addi $s6 $zero 2
65 mul $s7 $s7 $s6
66 sw $s7 20($sp)
67 sw $s7 24($sp)
68 j Label3
69 Label5:
70 lw $v0 20($sp)
71 lw $ra 4($sp)
72 jr $ra
73 end:
```

图 6 代码



```
Go: running mips1.asm

Go: execution terminated by null instruction.

Clear

Assemble: assembling C:\Users\youmeng\Desktop\MARS\mips1.asm

Assemble: operation completed successfully.
```

图 7 编译

				Registers		
				Coproc 1		
				Coproc 0		
				Name	Number	Value
				\$zero	0	0x00000000
				\$at	1	0x00000001
				\$v0	2	0x000000c0
				\$v1	3	0x00000000
				\$a0	4	0x00000000
				\$a1	5	0x00000000
				\$a2	6	0x00000000
				\$a3	7	0x00000000
				\$t0	8	0x00000000
				\$t1	9	0x00000000
				\$t2	10	0x00000000
				\$t3	11	0x00000000
				\$t4	12	0x00000000
				\$t5	13	0x00000000
				\$t6	14	0x00000000
				\$t7	15	0x00000000
				\$s0	16	0x00000000
				\$s1	17	0x00000000
				\$s2	18	0x00000000
				\$s3	19	0x00000003
				\$s4	20	0x0000000c
				\$s5	21	0x00000008
				\$s6	22	0x00000064
				\$s7	23	0x000000c0
				\$t8	24	0x00000000
				\$t9	25	0x00000000
				\$k0	26	0x00000000
				\$k1	27	0x00000000
				\$gp	28	0x10008000
				\$sp	29	0x10010000
				\$fp	30	0x00000000
				\$ra	31	0x00400074
				pc		0x00400100
				hi		0x00000000
				lo		0x000000c0

图 8 执行

对于错误测试 TestWrong2.txt，输出 “error，未定义的变量 f”：

```

22 int main()
23 {
24     int a;
25     int b;
26     int c;
27     int d;
28
29     a=3;
30     b=4;
31     c=max(a,b);
32     d=sum(a,b);
33
34     f=min(c,d);
35     return 0;
36 }
37 $

```

问题 输出 调试控制台 终端

```

PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> ./main
error:语法错误: 第34行, 变量f未声明
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> g++ -c main.cpp common.cpp myParse.cpp grammarParse.cpp
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> g++ main.o common.o myParse.o grammarParse.o -o main.exe
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> ./main
error:语法错误: 第32行, 函数min参数不匹配
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> g++ -c main.cpp common.cpp myParse.cpp grammarParse.cpp
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> g++ main.o common.o myParse.o grammarParse.o -o main.exe
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> ./main
error:语法错误: 第34行, 变量f未声明
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler>

```

图 9 未定义的变量

对于错误测试 TestWrong3.txt, 输出 “error, min 函数参数不匹配”:

```
21 int main()
22 {
23     int a;
24     int b;
25     int c;
26     int d;
27     int f;
28     a=3;
29     b=4;
30     c=max(a,b);
31     d=sum(a,b);
32     f=min(a,c,d);
33     return 0;
34 }
```

问题 输出 调试控制台 终端

PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> ./main
error:语法错误: 第34行, 变量f未声明
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> g++ -c main.cpp common.cpp myParse.cpp grammarParse.cpp
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> g++ main.o common.o myParse.o grammarParse.o -o main.exe
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> ./main
error:语法错误: 第32行, 函数min参数不匹配
PS C:\Users\youmeng\Desktop\study\thefiveterm\fundamentalsOfCompiling\yuzhuoCompiler> |

图 10 函数参数不匹配

5.2 时间复杂度分析

1. 在词法分析中, 对于整体调用函数 work, 时间复杂度最大的函数为 NFAconvert () 函数。在 NFA 转化成 DFA 的过程中, 需要不断的进行闭包操作。不考虑额外的消耗时间, 在外层的 while 循环下, 当输入串的长度为 k 时, 对于下面一级, 设所有状态总共有 m 个出口, 则单独考虑递归的部分对应计算 \square 闭包复杂度为 $O(m)$, 而设状态的数量为 n, 则对应非递归的部分时间复杂度为 $O(n)$, 综上时间复杂度为 $O(k(m+n))$ 。
2. 在语法分析中, $_work()$, 其中建立项目集的闭包, 其复杂度最高。从状态 I0 开始, 当非终结符与终结符数量和为 m, n 时, I0 便可能会有 $m+n-1$ 个出口, 最多可产生新的 $m+n-1$ 中状态。在最坏的情况下, 每个状态产生 $m+n-1$ 中状态, 但这些状态一般都是重复的, 因为可能已经产生的状态。只有非终结符优先转态的组合。接着考虑每一个产生式的长度, 设为 k, 产生式的每一个位置均可产生一种新的状态。m 个非终结符, 最多有 m 个产生式, 且每个产生式的长度为最多为 k, 对每一个状态, 都要进行 $O(n)$ 的递归。则时间复杂度为 $O(n*m2k2)$;
3. 在目标代码生成部分, handleBlock() 获得活跃变量的时间复杂度较高, 因为它需要从函数当前的 Block 递归地向后寻找, 是否存在出口活跃的变量, 每个递归调用内不需要遍历后面的基本块的每一条四元式, 判断其左、右操作数是否是本块中的目的操作数, 如果是则活跃, 如果都不是, 则非活跃。每个块有两个出口, 假设有 m 个函数, 每个函数有 n 个 Block, 每个 Block 有 k 条四元式, 那么时间复杂度为 $O(m*2n*k)$,

假设函数内划分的基本块很多 (n 很大), 则获得活跃变量的复杂度是相当大的。

5.3 模块设计时的思考

1. 在词法分析器设计时, 为什么要单独构造识别标识符的 DFA? 实际上, 在逐个字符扫描源程序时, 得到一个词后, 可以直接判断是否满足标识符命名规则, 而不必大费周章地推导标识符满足的文法、构造 NFA、NFA 转化为 DFA、用 DFA 来识别该标识符。但考虑到, 标识符和常量的定义应该是由某个文法来决定的, 为了正规化判别过程, 采用构造 DFA 识别标识符的方式。
2. 在语法分析器设计时, 为什么要将文法中的每个符号定义为一个子类, 继承符号类 `_Symbol`? 虽然在很多子类中, 数据成员是相同的, 我们完全可以使用结构体来代替, 这样就可以减少许多相似的结构定义。但为了问题梳理时方便, 选择将每一个符号定义为一个子类, 在每一种 case 分析时清晰明了, 也符合面向对象的编程思路。
3. 在目标代码生成时, 待用/活跃信息如何记录? 表如何设计? 采用什么样的数据结构存储呢? 考虑到生成目标代码是以函数为单位进行的, 而待用/活跃信息是以基本块为单位获得的, 具体到每一条四元式的操作数, 都要体现出待用/活跃情况, 从这个角度考虑, 将活跃信息定义在四元式上, 即 `QuaternaryWithInfo`, 里面包含一条四元式、左操作数活跃信息、右操作数活跃信息, 一个基本块内有 n 条四元式, 因此 `QuaternaryWithInfo` 定义在一个 `BlockWithInfo` 中, 一个函数又有多个 `BlockWithInfo`, 因此用一个 `Vector` 来存储。最后, 一个程序可能会有多个函数, 函数名不重复, 于是想到使用 `map` 对函数名做一个 `vector<BlockWithInfo>` 的映射 (这里实际上简化了, 如果考虑函数重载, 映射应加上函数返回值和参数列表加以区分), 最终得到了 `map<string,vector<BlockWithInfo>> funcBlocksWithInfo` 这个结构, 存储程序中所有的待用/活跃信息。
4. 目标代码生成中, 如何来进行寄存器分配, 使得程序不必频繁生成存数取数指令, 降低访存次数, 加快程序运行效率呢? 这里应按照老师上课讲的“尽可能留”、“尽可能用”、“尽量腾空”的思想优化寄存器分配, 提升程序运行效率。那么还可以再哪些地方做优化, 提升编译效率呢? 可以在输出中间代码后, 按照等价原则、有效原则和合算原则, 将源程序划分为若干基本块, 考虑删除公共子表达式、代码外提、强度削弱、变换循环控制条件、合并已知量、复写传播、删除无用赋值等方式, 输出优化后的四元式序列, 再将这个优化后的四元式序列交给目标代码生成器, 生成目标代码。

5.4 调试时遇到的问题及解决方法

1. 得到的 First 集和预期不符。

```

skyoung@skyoung-virtual-machine:~/parse/work$ g++ -o main main.cpp myParse.cpp grammarParse.cpp common.cpp
skyoung@skyoung-virtual-machine:~/parse/work$ ./main
26
非终结符:L
epsilon
非终结符:M
epsilon
非终结符:N
epsilon
非终结符:P
!= ( ) * + , - / ; < <= == > >= ID NUM else epsilon if int return void while { }
非终结符:argument_list
!= ( ) * + , - / < <= == > >= ID NUM epsilon
非终结符:assign_sentence
!= ( ) * + , - / ; < <= == > >= ID NUM epsilon
非终结符:declare_block
!= ( ) * + , - / ; < <= == > >= ID NUM else epsilon if int return void while { }
非终结符:declare_expression
!= ( ) * + , - / ; < <= == > >= ID NUM else epsilon if int return void while { }
非终结符:declare_fun
!= ( ) * + , - / ; < <= == > >= ID NUM else epsilon if int return void while { }
非终结符:declare_var
; epsilon
非终结符:fun_parameter
, ID epsilon int void
非终结符:if_sentence
!= ( ) * + , - / ; < <= == > >= ID NUM else epsilon if return while { }
非终结符:param
ID epsilon int
非终结符:parameter_list
, ID epsilon int
非终结符:partial_declare
epsilon
非终结符:partial_declare_var
ID epsilon int
非终结符:partial_declarer
; ID epsilon int
非终结符:return_sentence
!= ( ) * + , - / ; < <= == > >= ID NUM epsilon return
非终结符:sentence
!= ( ) * + , - / ; < <= == > >= ID NUM else epsilon if return while { }
非终结符:sentence_block
!= ( ) * + , - / ; < <= == > >= ID NUM else epsilon if return while { }
非终结符:sentence_list
!= ( ) * + , - / ; < <= == > >= ID NUM else epsilon if return while { }
非终结符:sub_expression
!= ( ) * + , - / < <= == > >= ID NUM epsilon
非终结符:subsub_expression
!= ( ) * + , - / < <= == > >= ID NUM epsilon
非终结符:value
!= ( ) * + , - / < <= == > >= ID NUM epsilon
非终结符:value_expression
!= ( ) * + , - / < <= == > >= ID NUM epsilon
非终结符:while_sentence
!= ( ) * + , - / ; < <= == > >= ID NUM else epsilon if return while { }

```

分析代码发现：

```

if (_nowSymbol._isTs)
{
    //是终结符，直接加入

    if (_firstSet[_leftSymbol].count(_nowSymbol) == 0)
    {
        _firstSet[_leftSymbol].insert(_nowSymbol);
        _finishFlag = 0;
        break;
    }
}

```



```

if (_nowSymbol._isTs)
{
    //是终结符，直接加入

    if (_firstSet[_leftSymbol].count(_nowSymbol) == 0)
    {
        _firstSet[_leftSymbol].insert(_nowSymbol);
        _finishFlag = 0;
    }
    break;
}

```

这里在遍历产生式右部符号的时候，如果右边符号是终结符就应该直接退出。但前面的写法如果遇到已经在 first 集中的符号是不会退出的。

```

skyyoung@skyyoung-virtual-machine:~/parse/work$ g++ -o main main.cpp myParse.cpp grammarParse.cpp common.cpp
skyyoung@skyyoung-virtual-machine:~/parse/work$ ./main
非终结符:L
epsilon
非终结符:M
epsilon
非终结符:N
epsilon
非终结符:P
int void
非终结符:argument_list
( ID NUM epsilon
非终结符:assign_sentence
ID
非终结符:declare_block
int void
非终结符:declare_expression
int void
非终结符:declare_fun
(
非终结符:declare_var
;
非终结符:fun_parameter
int void
非终结符:if_sentence
if
非终结符:param
int
非终结符:parameter_list
int
非终结符:partial_declare
epsilon
非终结符:partial_declare_var
int
非终结符:partial_declarer
int
非终结符:return_sentence
return
非终结符:sentence
ID if return while
非终结符:sentence_block
{
非终结符:sentence_list
ID if return while
非终结符:sub_expression
( ID NUM
非终结符:subsub_expression
( ID NUM
非终结符:value
( ID NUM
非终结符:value_expression
( ID NUM
非终结符:while_sentence
while

```

2. 第一条指令的跳转有误。

1 (j,_,_,-1)

忘记在最后一步回填，对 main 进行回填：

```
case 1:
{
    // P -> N declare_block
    _Fun *_f = _LookUpFunc("main");
    _PopStack();
    _N *_n = (_N *)_PopStack();
    _code._BackPatch(_n->_nextList, _f->_enterPoint);
    _Symbol *_p = new _Symbol(_nowPro._left);
    _PushStack(_p);
    break;
}
```

3. demo 函数划分基本块错误：在检查划分的基本块时，发现 demo 这个函数中并没有基本块和四元式，显然是错误的。

```

funcblock.txt
1  [demo]
2  [program]
3  program:
4      (get,_,_,a)
5      (get,_,_,b)
6      (get,_,_,c)
7      (=,0,_,i)
8      (+,b,c,T0)
9      (j<=,a,T0,Label2)
10     next1 = 1
11     next2 = 2
12     Label1:
13         (*,b,c,T1)
14         (+,T1,1,T2)
15         (+,a,T2,T3)
16         (=,T3,_,j)
17         (j,_,_,Label3)
18         next1 = 3
19         next2 = -1
20     Label2:
21         (=,a,_,j)
22         next1 = 3
23         next2 = -1
24     Label3:
25         (j>,i,100,)

```

经过调试发现，错误如下：

```

71 //映射关系: {上一个块, {当前块的名称, 当前块的块号}}
72 map<int, pair<string,int>> trans;
73
74 int enter, lastEnter = blockEnters.top();
75 blockEnters.pop();
76
77 bool isFirstBlock = true;
78
79 while(blockEnters.size())
80 {
81     enter = blockEnters.top();
82     blockEnters.pop();
83
84     //这里实际上处理两个跳转到同一出口的情况
85     if(enter == lastEnter) continue;
86
87     //将两个入口点之间的所有四元式放入
88     for(int i = lastEnter; i != enter; i++)
89         block.quaters.push_back(interMediateCode[i]);
90
91     if(isFirstBlock)
92     {
93         block.name = curFunName;
94         isFirstBlock = false;
95     }

```

若pop后为空, 不会进入循环将四元式放到容器中, 只有一个基本块的情况就会出错

于是在 BlockEnter 中压入一个结束块的入口点, 统一进行处理:

```

//针对函数只有一个基本块的情况, 压入一个结束标志方便统一处理
blockEnters.push((iter + 1)->first);
while(blockEnters.size())
{
    enter = blockEnters.top();
    blockEnters.pop();

    //这里实际上处理两个跳转到同一出口的情况
    if(enter == lastEnter) continue;

    //将两个入口点之间的所有四元式放入
    for(int i = lastEnter; i != enter; i++)
        block.quaters.push_back(interMediateCode[i]);

    if(isFirstBlock)
    {
        block.name = curFunName;
        isFirstBlock = false;
    }
    else
        block.name = label - MakeLabel();
}

```

于是问题解决：

```
funcblock.txt
1  [demo]
2  demo:
3      (get,_,_,a)
4      (+,a,2,T5)
5      (=,T5,_,a)
6      (*,a,2,T6)
7      (return,T6,_,_)
8      next1 = -1
9      next2 = -1
10 [main]
11 main:
12     (=,3,_,a)
13     (=,4,_,b)
14     (=,2,_,c)
15     (para,c,_,_)
16     (call,demo,_,_)
17     next1 = 1
18     next2 = -1
19 Label6:
20     (=,RETURN,_,T7)
21     (para,a,_,_)
22     (para,b,_,_)
23     (para,T7,_,_)
24     (call,program,_,_)
25     next1 = 2
26     next2 = -1
27 Label7:
```

4. 待用/活跃信息求取出错。

所有左右操作数的待用/活跃信息在初始时均被置为非待用非活跃，开始并没有意识到这个问题，原因是对待用/活跃信息的含义掌握和理解不清，只是照本宣科地阅读ppt上的算法。后来发现寄存器分配时始终无法按照预想的”三个原则“来进行，回过头来分析才发现所有左右操作数的待用/活跃信息初始化错误，虽然待用信息确实是初始化为-1，但是否活跃，需要根据后面基本块是否出现该变量来判断。这里主要是对文字阅读的理解偏差造成的，导致了巨大的返工，花费了较多的时间。这里再次提醒自己，计算机科学真的是一门逻辑非常缜密的学科，稍微不小心，就很可

能得到错误答案。

另外，在求取得用/活跃信息时，还出现了大量的待用信息为 0（只被第一条语句待用），这显然是不合理的，因为分析待用活跃信息时，是从后向前分析的，只有可能是后面的四元式中的变量被前面的待用，不可能出现前面的被后面的待用。

```
[program]
program:
(get,_,_,a) 左操作数: (^,^) 右操作数: (^,^) 目的操作数:(5,y)
(get,_,_,b) 左操作数: (^,^) 右操作数: (^,^) 目的操作数:(4,y)
(get,_,_,c) 左操作数: (^,^) 右操作数: (^,^) 目的操作数:(4,y)
(=,0,_,i) 左操作数: (^,^) 右操作数: (^,^) 目的操作数:(^,^)
(+,b,c,T0) 左操作数: (0,^) 右操作数: (0,^) 目的操作数:(5,y)
(j<=,a,T0,Label12) 左操作数: (0,^) 右操作数: (^,^) 目的操作数:(^,^)
    next1 = 1
    next2 = 2
```

这里是没有对左右操作数初始化造成的，因为 map 在未初始化时取值结果就是 0。

究其原因，还是对待用/活跃信息含义理解不清导致的。

5. 目标代码中，寄存器名称出现诡异字符：

```
lui $sp, 0x1001
j main
demo:
sw $ra 4($sp)
lw Regs: 8($sp)
addi eRegs: $zero 2
add eeRegs: Regs: eRegs:
lw reeRegs: 0($sp)
```

```
main:
addi $s7 $zero 3
addi $s6 $zero 4
addi $s5 $zero 2
sw $s6 8($sp)
sw $s5di $sp $sp
sw $sp 12($sp)
addi $sp $sp 12
jal demo
lw $sp 0($sp)
```

调试发现，是下面的语句造成的：

```

void ObjectCodeGen::genForFuncBlock(int curBlockIdx)
{
    //初始化空闲寄存器
    freeRegs.clear();
    for(int i = 0; i <= 7; i++)
        freeRegs.push_back("$s" + (i));
    objectCodes.push_back(curIBlock->name + ":");
}

```

未强转, 前面的"\$s"默认未char*, 并不能用string的的+重载运

这里是对 c++ 的语言特性掌握不深导致的, 对 i 加上 to_string 强转为 string 类型后, 再去与前面的 char* 去相加, 才会调用 string 重载的 + 运算符, 进行字符串拼接。

6. 存数指令生成出错:

```

Label3:
    (j>,i,100,Label15) 左操作数: (^,y) 右操作数: (^,^) 目的操作数:(^,^)
    next1 = 4
    next2 = 5
Label4:
    (*,j,2,T4) 左操作数: (^,^) 右操作数: (^,^) 目的操作数:(1,y)
    (=,T4,_,i) 左操作数: (^,y) 右操作数: (^,^) 目的操作数:(2,y)
    (=,i,_,j) 左操作数: (^,y) 右操作数: (^,^) 目的操作数:(^,y)
    (j,_,_,Label13) 左操作数: (^,^) 右操作数: (^,^) 目的操作数:(^,^)
    next1 = 3
    next2 = -1
Label5:
    (return,i,_,_) 左操作数: (^,^) 右操作数: (^,^) 目的操作数:(^,^)
    next1 = -1
    next2 = -1

```

```

Label3:
lw $s7 20($sp)
addi $s6 $zero 100
bgt $s7 $s6 Label15
Label4:
lw $s7 24($sp)
addi $s6 $zero 2
mul $s7 $s7 $s6
sw $s7 24($sp)
j Label13
Label5:
lw $v0 20($sp)
lw $ra 4($sp)
jr $ra
end:

```

只存了一个 j, i 没有存, 下次循环马上又要取 i, 但因为没有存, 所以取到的还是原

来的 i ，这显然是不合理的。

这个地方也花费了大量时间来调试，虽然通过打断点的方式，很快发现了是生成存数指令策略和寄存器分配策略不拟合导致的，但思考解决方法，花费了大量的时间。生成存数指令的策略是：如果某个变量在出了基本块之后是活跃的，并且它的值不在内存中而只在寄存器中，需要生成存数指令。而在给 AVALUE 这个寄存器赋初值时，先将所有出口活跃变量都存在了内存里 (AVALUE[a].insert(a))，这样本基本块内直接使用时，才知道去哪里中找该变量。但实际上，这一步应该是将当前基本块的所有入口活跃变量初始化到内存里，而不是上一个块的出口活跃变量（因为上一个块的出口有两个，可能包含又另一个块的入口活跃变量），这导致 i 先被存到了内存里，与生成存数指令冲突，故在此处 i 没有生成存数指令。

这里看出，虽然课本和 ppt 中并未提及到入口活跃变量，但在具体分析问题时，有了这个东西，处理起来就会方便很多，所以在实际应用时也不能照搬理论，要灵活处理。

最后引入入口活跃变量后进行处理，在此处生成了两条存数指令，结果正确。

```
Label13:
lw $s7 20($sp)
addi $s6 $zero 100
bgt $s7 $s6 Label15
Label14:
lw $s7 24($sp)
addi $s6 $zero 2
mul $s7 $s7 $s6
sw $s7 20($sp)
sw $s7 24($sp)
j Label13
```


六、用户使用说明

6.1 编译项目

6.1.1 cmake 打包生成 makefile

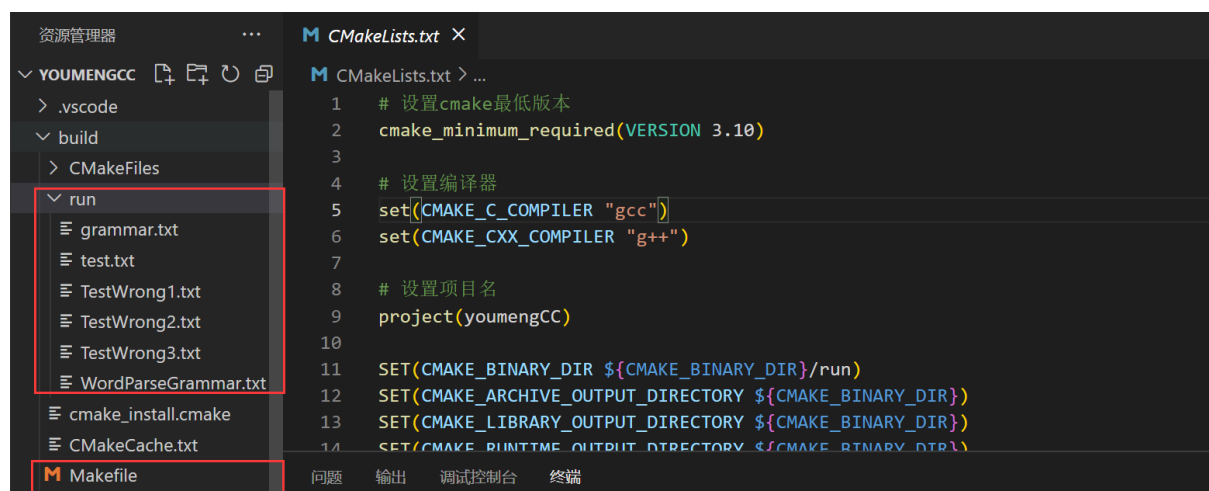
在源代码目录，打开命令行，新建 build 文件夹，切换到 build 目录，用 cmake 工具将工程所需文件打包到 run 文件夹下，并生成 makefile。

```
PS D:\totalStudy\sixth\Compiler\youmengCC> mkdir build

目录: D:\totalStudy\sixth\Compiler\youmengCC

Mode                LastWriteTime         Length Name
----                -
d-----          2023/5/19      10:48             build

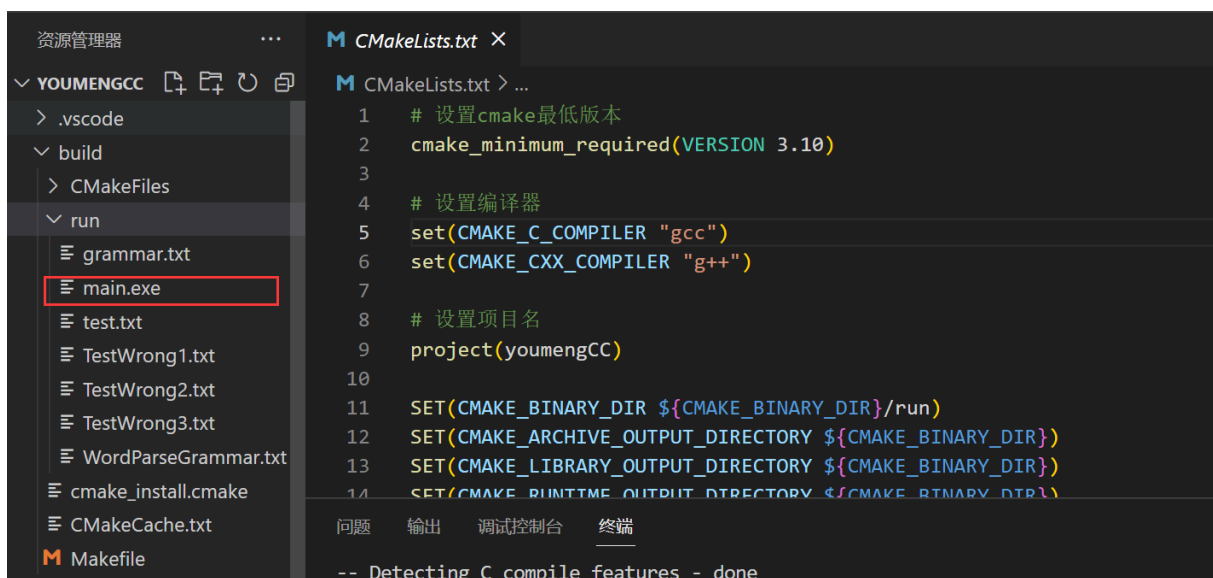
PS D:\totalStudy\sixth\Compiler\youmengCC> cd build
PS D:\totalStudy\sixth\Compiler\youmengCC\build> cmake -G "Unix Makefiles" ..
-- The C compiler identification is GNU 9.2.0
-- The CXX compiler identification is GNU 9.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: C:/MinGW/bin/gcc.exe - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: C:/MinGW/bin/g++.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: D:/totalStudy/sixth/Compiler/youmengCC/build
PS D:\totalStudy\sixth\Compiler\youmengCC\build> |
```



6.1.2 编译

执行 make 指令，编译项目，将在 run 文件下生成 main.exe

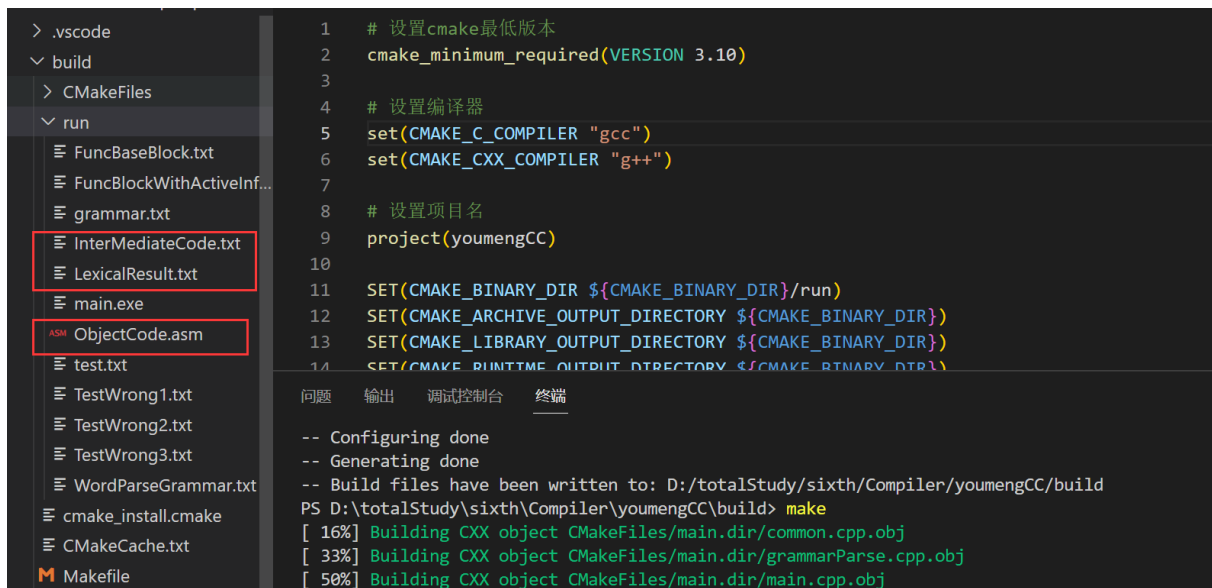
```
PS D:\totalStudy\sixth\Compiler\youmengCC\build> make
[ 16%] Building CXX object CMakeFiles/main.dir/common.cpp.obj
[ 33%] Building CXX object CMakeFiles/main.dir/grammarParse.cpp.obj
[ 50%] Building CXX object CMakeFiles/main.dir/main.cpp.obj
[ 66%] Building CXX object CMakeFiles/main.dir/myParse.cpp.obj
[ 83%] Building CXX object CMakeFiles/main.dir/ObjectCodeGen.cpp.obj
[100%] Linking CXX executable run/main.exe
[100%] Built target main
```



6.2 程序运行

到 run 文件夹运行 main.exe，程序将会提示输入要编译的源文件名称，项目提供 4 个测试用例，1 个编译正确的 test.txt 和 3 个编译错误的 TestWrong.txt。编译正确的，将会输出词法分析结果、中间代码和目标代码，编译错误的，控制台会给出相关提示：

```
PS D:\totalStudy\sixth\Compiler\youmengCC\build> cd run
PS D:\totalStudy\sixth\Compiler\youmengCC\build\run> ./main
Please input the testfileName:
test.txt
Compile successfully! Please open the output file to check the result!
请按任意键继续. . .
```



```

PS D:\totalStudy\sixth\Compiler\youmengCC\build\run> ./main
Please input the testfileName:
TestWrong1.txt
error:归约失败,找不到对应的action
PS D:\totalStudy\sixth\Compiler\youmengCC\build\run> ./main
Please input the testfileName:
TestWrong2.txt
error:语法错误: 第34行, 变量f未声明
PS D:\totalStudy\sixth\Compiler\youmengCC\build\run> ./main
Please input the testfileName:
TestWrong3.txt
error:语法错误: 第31行, 函数min参数不匹配

```

七、总结与收获

7.1 总结

通过《编译原理》这门课程，走通了程序编译的整个过程，现将其总结如下：

7.1.1 词法分析

1. 根据语言特性划分词的种类，标识符、常量、数字一般按照类别划分，关键字按照一词一种的方式划分比较方便；
2. 扫描源程序，逐一分析每一次，一般只需向前搜索一步，如果多读取了字符，需要回退操作；
3. 可根据标识符命名的文法构造 DFA 来识别标识符是否合法，这一项是可选内容，也可以直接在程序中规定符合哪些特点的符号为标识符；
4. 输出是 < 内容，种别 > 键值对序列。

7.1.2 语法语义分析

1. 需要先设计语言的文法，以产生式为单位读入程序中；
2. 根据语言的文法构造 DFA，用 DFA 去识别词法分析的输出，是否符合该语言的语法规则，该过程需要加上语法制导翻译，同时对语义进行分析；
3. 输出函数表和中间代码

7.1.3 优化

该部分是可选内容，是为了使得目标代码生成更加简洁，提高程序执行效率。优化可分为局部优化、循环优化、全局优化三个级别，常见的优化方法有：删除多余运算(或称删除公用子表达式)、代码外提、强度削弱、变换循环控制条件、合并已知量、复写传播、删除无用赋值等等。

7.1.4 目标代码生成

1. 拿到函数表和中间代码后，划分基本块；
2. 求各基本块的出口/入口活跃变量；
3. 根据四元式和汇编知识生成相应的汇编代码，大致可分为赋值、运算、函数跳转、参数传递、函数返回、控制语句几类目标代码的生成，按照几种操作的特点，结合汇编知识，生成目标代码即可。
4. 最终输出可以汇编执行的目标代码

7.2 体会与收获

1. 经过两个学期的学习和努力，终于将整个编译流程跑通，感触最深的是一个道理：做简单的事收益不大，做困难的事收获良多。当然，在未知和迷茫之中挣扎是极其痛苦的一件事，但是人往往是在痛苦中成长的。要想真正学到知识，就应该用于去探索未知的领域，而不是待在自己的舒适圈。尤其对于我们计算机专业的学生来说，技术的发展迭代是日新月异的，我们需要保持良好的学习习惯，“活到老，学到老”，才能尽可能避免淹没在时代汹涌的发展潮流之中。当然，困难的事之所以困难，还有一点是在于，并不是说你去做，就一定能成功，需要掌握正确的学习、研究方式，如何向老师请教、如何与同学交流沟通、参考别人的设计并化为己用，都是可行的方法。切忌个人死磕，古人说“独学而无友，则孤陋寡闻”是有道理的。
2. 良好的数据结构设计是提高代码效率的关键。由于上学期是组队完成实验，自己写的词法分析器、语法分析器因数据结构不合理、代码糟糕等原因，队友未采用并重写了代码。本次实验也是在他写完中间代码生成的基础上完成的，通过阅读他的代码，我学习到了很多新的东西。一是代码的整洁和规范程度，确实整洁、清晰，易于阅读；二是数据结构的设计非常合理，使得某些处理变得简洁清晰；三是对于 C++ 内置 STL 的使用，他的代码中使用了大量的 `map`、`list`、`vector`、`set` 来存储相关信息，由于自己对这些 STL 的操作不熟练，是不敢拿来用的。但不用，不就一直不会熟练吗？于是借着这个契机，也试着使用这些 STL 去完成后面的设计，更加熟练了对 c++ 语言特性的掌握。
3. 代码实现和伪代码描述差别很大。不要以为掌握了算法流程就万事大吉，归根结底需要了解并熟悉算法的原理，在一些代码的细节处理上才能准确无误，实现正确的功能。比如本次实验在待用/活跃信息和寄存器分配策略那块的设计，参考 ppt 上的算法，因为没有理解透彻，很多细节没有注意到，导致生成了错误待用/活跃信息和分配的寄存器不理想，需要消耗大量时间去返工。所以理解原理是很重要的。
4. 在完成整个实验的过程中，既是对课堂所学到知识方法的一次学习与巩固；又是一次宝贵的设计实践活动。在整个过程中，我和上学期的小组成员展开了许多讨论与设计方法的设计，最终在语法分析的基础上完成了本次实验。尽管编译过程并未做过多的优化，也只能识别 c 语言的部分特性，但对于我了解编译原理、设计实现编译程序都是一次宝贵的经历与体验。它使得设计和实现编译程序不再是我想象中的那样是一件十分困难、令人生畏的事。