Java 子语言语法语义分析器

王韬懿

08111302

1120132046

目 录

一 设计思路

1.1 语法分析设计思路

在语法分析中采用了递归下降法对源程序进行分析，对于一个经过词法分析之后的程序，在语法分析器中，以这个语法分析器作为中心，来调用词法分析的功能得到一个单词。然后基于递归下降的思想逐个的对语句进行分析，最后形成一个语法树。

以我的测试程序为例：

```
int numA = 1;
int numB = 2;
int sum = 0;
while (sum < 100)
    sum = sum + numA * numB;
```

在分析这样的源程序的时候，会经过下面的一些函数的遍历：

```
/* 整个程序语句 */
TreeNode * Parser::programStmt() {
    /* 这里简化为直接在main函数里面执行 */
    token = getToken();
    TreeNode * tempNode = mulSentenceStmt();
    return tempNode;
}

/* 多语句 */
TreeNode * Parser::mulSentenceStmt() {
    TreeNode * str = NULL;
    TreeNode * end = NULL;
    TreeNode * next = NULL;
    while (token == INT || token == CONST_INT8 || token == CONST_INT16 || token == ID ||
           token == WHILE || token == SEMICOLON || token == BRACKET_LL || token == BRACKET_LR) {
        next = sentenceStmt();
        /* 第一条语句 */
        if (str == NULL || end == NULL) {
            str = end = next;
        } else {
            end->sibling = next;
            end = next;
        }
    }

    return str;
}

/* 单个语句 */
TreeNode * Parser::sentenceStmt() {
    Parser::tokenList.clear();
    assignStart = false;
    if (tokenString == "while") {
        return whileStmt();
    } else {
        switch (token) {
            case INT:
            case CONST_INT8:
            case CONST_INT16:
            case ID:
                return assignStmt();
                break;

            default:
                return NULL;
                break;
        }
    }
    return NULL;
}
```

```
/* 赋值语句 */
TreeNode * Parser::assignStmt() {
    /* 过滤掉类型 */
    if (tokenString == "int") {
        token = getToken();
    }
    Parser::tokenList.clear();

    TreeNode *treeNode = new TreeNode;
    TreeNode &thisNode = *treeNode;
    thisNode.lineno = lineNumber;
    /* 连等尚未开始 */
    if (assignStart == false) {
        /* 初始化节点信息 */
        thisNode.nodeKind = STMTK;
        thisNode.stmtKind = ASSIGNK;
        thisNode.id = tokenString;
        token = getToken();
        match(ASSIGN);
        /* 开始赋值符号 */
        assignStart = true;
        token = getToken();
        /* 多项式运算里面可能出现的符号 */
        if (token == ID || token == CONST_INT || token == CONST_INT8 || token == CONST_INT16
            || token == ADD || token == MINUS || token == MUL || token == DIV || token == BRACKET_LL || token ==
                BRACKET_LR) {
            thisNode.child.clear();
            thisNode.child.push_back(assignStmt());
        } else {

        }
    } else {
        /* 开始连等运算 */
        Parser::tokenList.push_back({token, tokenString});
        thisNode.id = tokenString;
        /* 如果连等继续 */
        if (token == ASSIGN && (tokenList.back().type == ID)) {
            tokenList.clear();
            token = getToken();
            thisNode.nodeKind = STMTK;
            thisNode.stmtKind = ASSIGNK;
            thisNode.child.clear();
            thisNode.child.push_back(assignStmt());
        } else {
            /* 最后一个是表达式 */
            delete (treeNode);
            TreeNode * tmpExp = expStmt();
            match(SEMICOLON);
            token = getToken();

            return tmpExp;
        }
    }

/* while语句 */
TreeNode * Parser::whileStmt() {

    TreeNode * treeNode = new TreeNode;
    TreeNode & thisNode = *treeNode;
    thisNode.lineno = lineNumber;
    thisNode.nodeKind = STMTK;
    thisNode.stmtKind = WHILEK;
    thisNode.child.clear();
    // 匹配while语句
    token = getToken();
    match(BRACKET_SL);
    token = getToken();
    // ()中的表达式语句
    Parser::tokenList.clear();
    thisNode.child.push_back(expStmt());
    match(BRACKET_SR);
    token = getToken();
    // 暂时处理单个赋值语句
    thisNode.child.push_back(assignStmt());

    return treeNode;
}
```

之后再经过语法分析器之后会得到一颗语法树，该语法树的结构如图 1 所示所示。

其中每一个节点都有以下几个属性：child, sibling, lineno, nodeKind, id 以及两个联合，如图 2 所示。

在语法树中，如果该节点代表的是一个语句，那么该节点的 sibling 就代表下一条语句，child 中存放与该节点相关的属性。

如果该节点是一个赋值语句，那么该节点的 ID 字段存放赋值语句等号左边的标志符，第一个孩子节点存放该赋值语句等号右边的值。

如果该节点是一个 while 语句，那么该节点的第一个孩子节点存放 while 语句中的判断表达式节点，第二个孩子节点存放 while 语句的循环体的部分对应的节点，就这样递归存储就得到了语法树。

在构建完毕这样一颗语法树之后，就可以根据这棵树来生成中间代码，这里我采用的是四元式代码。

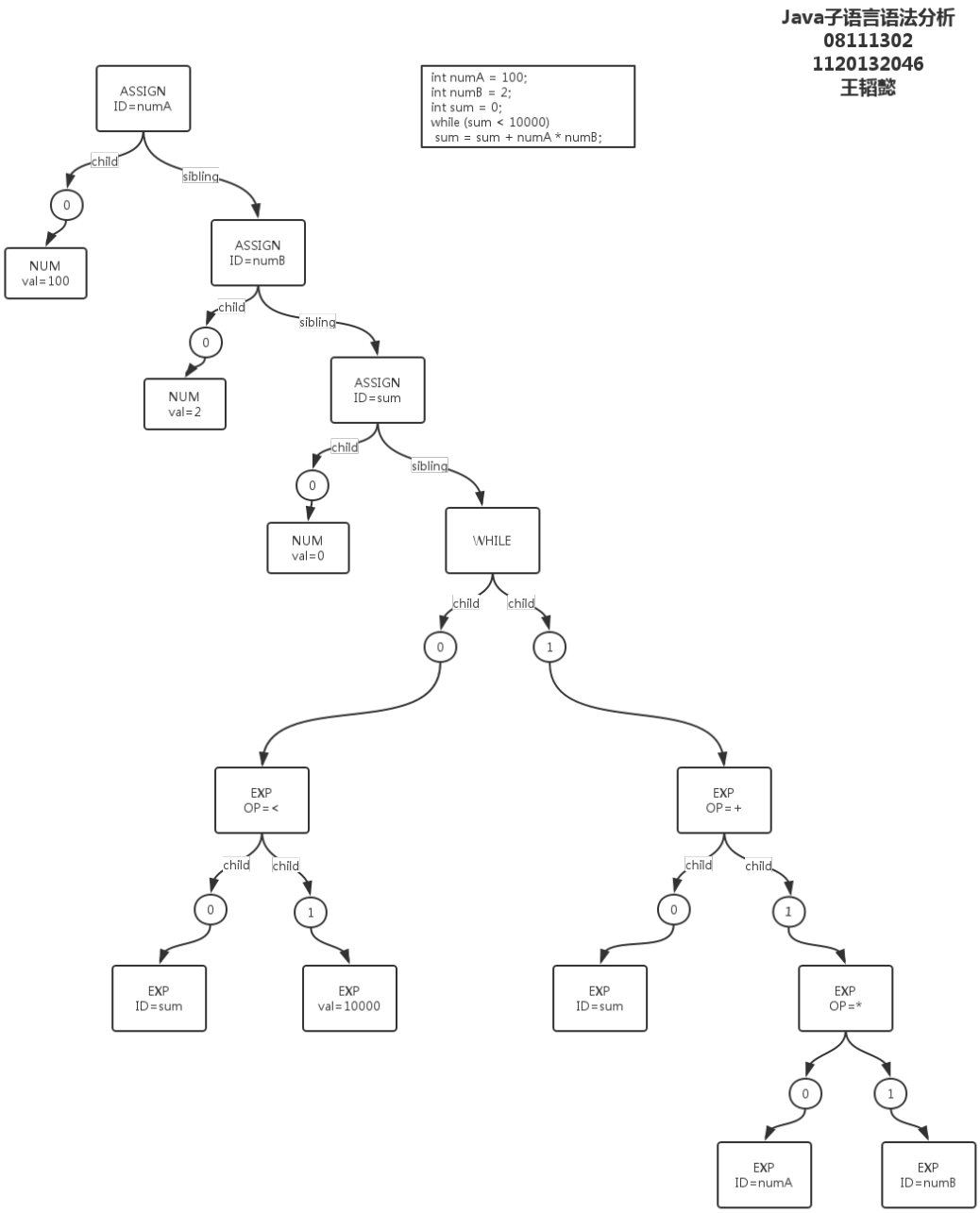


图 1 语法树结构图

```
/* 语法树节点 */
struct TreeNode {
public:
    TreeNode() {
        sibling = NULL;
        child.clear();
    }
    /* 孩子节点 */
    std::vector<TreeNode*> child;
    /* 兄弟节点 */
    TreeNode* sibling;
    /* 所在行号 */
    int lineno;
    /* 节点类型 */
    NodeKind nodeKind;
    /* 语句类型 */
    union {StmtKind stmtKind; ExpK expK;};
    /* 节点属性 */
    union {
        /* 操作符 */
        TokenType op;
        /* 整型常数 */
        int num;
    };
    /* 标志符名称 */
    std::string id;
};
```

图 2 语法树节点数据结构

1.2 四元式代码生成设计思路

在得到了这样一颗语法树之后，就可以递归遍历这个语法树来生成中间代码的四元式表示形式，首先看一下我的四元式的数据结构，如图 3 所示：

```
/* 四元式结构 */
typedef struct {
    /* 编号 */
    int no;
    /* 操作符 */
    std::string op;
    /* 参数1 */
    std::string arg1;
    /* 参数2 */
    std::string arg2;
    /* 结果 */
    std::string result;
    /* 以下均为辅助使用 */
    /* 需要回填的编号 */
    int backNo;
} Tuple4;
```

图 3 四元式结构

由于得到的语法树是采用递归下降分析方法实现的，因此在遍历这颗语法树的时候，根据每一个节点对应的不同属性，递归的对这颗语法树进行遍历，如图 4 所示。

```
/* 生成四元式中间代码 */
void Generator::codeGen(TreeNode *syntaxTree) {
    mulGen(syntaxTree);
}

/* 多个语句 */
void Generator::mulGen(TreeNode *syntaxTree) {
    while (syntaxTree != NULL) {
        sentenceGen(syntaxTree);
        syntaxTree = syntaxTree->sibling;
    }
}

/* 单个语句 */
void Generator::sentenceGen(TreeNode *syntaxTree) {
    if (syntaxTree->nodeKind != STMTK) {
        // TODO: handle error
        return;
    }
    switch (syntaxTree->stmtKind) {
        case ASSIGNK:
            assignGen(syntaxTree);
            break;
        case WHILEK:
            whileGen(syntaxTree);
            updateTuple4();
            break;
        default:
            // TODO: handle error
            break;
    }
}

/* while语句 */
void Generator::whileGen(TreeNode *syntaxTree) {
    /* 首先是括号之内的判断表达式 */
    expGen(syntaxTree->child[0]);
    /* 接着是判断结果之后的跳转，看其是否跳转到while语句的下一条语句 */
    int backNo = number - 1;
    Tuple4 tuple = newTuple4(number++, "jT", result, "", int2str(number + 1), 0);
    tuple4List.push_back(tuple);
    tuple = newTuple4(number++, "j", "", "", lastResult, 0);
    tuple4List.push_back(tuple);

    /* 然后执行循环体 */
    assignGen(syntaxTree->child[1]);

    /* 最后是循环执行，返回到条件判断部分 */
    tuple = newTuple4(number++, "j", "", "", int2str(backNo), backNo);
    tuple4List.push_back(tuple);
}
```

Unsequenced modification and access to 'number

```
/* 赋值语句 */
void Generator::assignGen(TreeNode *syntaxTree) {
    /* 赋值语句 */
    Tuple4 tuple;
    if (syntaxTree != NULL && syntaxTree->child.size() > 0) {
        if (syntaxTree->child[0]->nodeKind == STMTK && syntaxTree->child[0]->
            stmtKind == ASSIGNK) {
            /* 首先执行等号右边的表达式 */
            expGen(syntaxTree->child[0]);
            /* 然后生成最后一个赋值语句 */
            Tuple4 tuple = newTuple4(number++, "=", "last exe val no in assign", "",
                syntaxTree->id, 0);
            tuple4List.push_back(tuple);
            /* 表达式 */
        } else if (syntaxTree->child[0]->nodeKind == EXPK) {
            /* 后面直接跟着数字或者标志符 */
            if (syntaxTree->child[0]->stmtKind == NUMK || syntaxTree->child[0]->
                stmtKind == IDK) {
                tuple = newTuple4(number++, "=", syntaxTree->child[0]->id, "",
                    syntaxTree->id, 0);
                tuple4List.push_back(tuple);
                return;
            } else {
                expGen(syntaxTree->child[0]);
                /* 最后的赋值语句 */
                tuple = newTuple4(number++, "=", result, "", syntaxTree->id, 0);
                tuple4List.push_back(tuple);
            }
        }
    }
}
```

```
/* 表达式语句 */
void Generator::expGen(TreeNode *syntaxTree) {
    /* 操作符节点 */
    Tuple4 tuple;
    Lexer lexer;
    std::string arg1;
    std::string arg2;
    if (syntaxTree->expK == OPK) {
        /* 四元式中的第一个参数 */
        if (syntaxTree->child[0]->expK == NUMK || syntaxTree->child[0]->expK == IDK)
            {
            arg1 = syntaxTree->child[0]->id;
        } else {
            expGen(syntaxTree->child[0]);
            arg1 = result;
        }
        /* 四元式中的第二个参数 */
        if (syntaxTree->child[1]->expK == NUMK || syntaxTree->child[1]->expK == IDK)
            {
            arg2 = syntaxTree->child[1]->id;
        } else {
            expGen(syntaxTree->child[1]);
            arg2 = result;
        }
        switch (syntaxTree->op) {
            case EQU:
            case NE:
            case GT:
            case LT:
            case GE:
            case LE:
                /* 更新结果 */
                updateResultLabel(syntaxTree->op);
                /* 生成新的四元组 */
                tuple = newTuple4(number++, lexer.tokenMap[syntaxTree->op].first,
                    arg1, arg2, result, 0);
                tuple4List.push_back(tuple);
                break;
            case ADD:
            case MINUS:
            case MUL:
            case DIV:
                updateResultLabel(syntaxTree->op);
                /* 生成新的四元组 */
                tuple = newTuple4(number++, lexer.tokenMap[syntaxTree->op].first,
                    arg1, arg2, result, 0);
                tuple4List.push_back(tuple);
                break;
            default:
                break;
        }
    }
}
```

图 4 递归遍历生成四元式代码

遍历结束之后，所有的四元式都保存到 tuple4List 这个列表中，最后对这个列表遍历即可得到测试用例的四元式代码，如图 5 所示：

```
NO  OP ARG1 ARG2 RESULT
 1:( =, 100,     ,numA)
 2:( =,   2,     ,numB)
 3:( =,   0,     , sum)
 4:( <, sum,10000,  T1)
 5:(jT,  T1,     ,   7)
 6:( j,     ,    ,  11)
 7:( *,numA,numB,  T2)
 8:( +, sum,  T2,  T3)
 9:( =,  T3,     , sum)
10:( j,     ,    ,   4)
11:...
```

图 5 测试用例的四元式代码

二 源程序主要函数功能

2.1 工具类 util

```
/**
* 判断字符是否是字母
* @return 是:true, 否:false
*/
bool isAlpha(char c);

/**
* 判断字符是否是数字
* @return 是:true, 否:false
*/
bool idDigit(char c);

/**
* 判断字符是否是标识符号
* @return 是:true, 否:false
*/
bool isIdentifier(char c);

/**
* 判断字符是否是数值运算符
*/
bool isArithmeticOp(char c);

/**
* 判断字符是否是转义字符
*/
bool isESC(char c);

/**
 * 将数字转化为字符串
 */
std::string int2str(int val);
```

2.2 词法分析器类 Lexer

```
/* 词法分析器类 */
class Lexer {

public:
    /* 词法分析阶段程序错误标志 */
    int LEXER_ERROR;

    /* EOF结束标志 */
    int EOF_flag;

    /* 总单词个数 */
    int TOKEN_NUM;

    /* 输入文件流 */
    std::ifstream ifs;

    /* 每行单词个数统计 */
    std::map<int, int> lineTokenSumMap;

    /* 扫描出的所有单词 */
    std::vector<Token> tokenList;

    /* 错误信息列表 */
    std::vector<TokenErrorInfo> errList;

    /* 单词属性及其对应Token的关系 */
    std::map<TokenType, std::pair<std::string, std::string> > tokenMap;

    /**
     * 构造函数
     *
     * @fileName 文件名
     */
    Lexer(std::string fileName);

    /* 重载构造函数 */
    Lexer();

    ~Lexer();


    Lexer(const Lexer &) = delete;
    Lexer& operator=(const Lexer &) = delete;

    /**
     * 获得单词的token
     *
     * @return token状态
     */
    virtual TokenType getToken();
```

```
    /**
     * 运行词法分析器
     *
     * @param fileName 源代码文件
     * @param outFilename 词法分析结果输出文件
     */
    static void runLexer(std::string fileName, std::string outFileName);
protected:
    /* 扫描到的行数 */
    int lineNumber;

    /* 扫描到某行的位置 */
    int linePos;

    /* 每行单词的个数 */
    int lineTokenNum;

    /* 提取出来的单词 */
    std::string tokenString;

private:

    /* 读入的每一行字符流 */
    std::string lineBuf;

    /* DFA的状态 */
    DFAStateType currentState;

    /* 关键字及其对应Token的关系 */
    std::map<std::string, std::pair<TokenType, std::string> > keyWords;

    /* 界限符与对应Token的关系 */
    std::map<char, TokenType> delimeterMap;

    /**
     * 读取一行内容并存入lineBuf
     * @return void
     */
    void getOneLine();

    /**
     * 获得下一个字符
     *
     * @return 下一个字符
     */
    char getNextChar();
```

```
    /**
     * 回退一个字符
     *
     * @return void
     */
    void ungetNextChar();

    /**
     * 扫描错误
     *
     * @return void
     */
    void scanError();

    /**
     * 打印token的信息
     *
     * @param token token类型
     * @param tokenString token 保存字符串
     */
    void printToken(TokenType token, std::string tokenString);

    /**
     * 创建一个token
     *
     * @param type token类型
     * @param tokenString token 保存的字符串
     * @return 新的token
     */
    Token createToken(TokenType type, std::string tokenString);

    /**
     * 创建一个token出错的错误信息
     *
     * @param errorToken 错误的单词
     * @return 错误信息节点
     */
    TokenErrorInfo createTokenErrorInfo(std::string errorToken);

    /**
     * 获得一个token的类型名称
     *
     * @param type token 类型
     * @param tokenString token的值
     */
    std::string getTokenTypeName(TokenType type, std::string tokenString);
};
```

2.3 语法分析器类 Parser

```
class Parser : public Lexer {
public:

    /**
     * 构造函数
     *
     * @param fileName 扫描文件名
     */
    Parser(std::string fileName):
        Lexer(fileName), tokenList({}), assignStart(false), treeRoot(NULL){};

    /* 析构函数 */
    ~Parser();

    /**
     * 生成语法分析树
     *
     * @return 语法树
     **/
    TreeNode* parse();
private:

    /**
     * 整个程序的树
     *
     * @return 树的根节点
     */
    TreeNode * programStmt();

    /**
     * 多重语句构建的树
     *
     * @return 树的根节点
     */
    TreeNode * mulSentenceStmt();

    /**
     * 单条语句构建的树
     *
     * @return 树的根节点
     */
    TreeNode * sentenceStmt();

    /**
     * 赋值语句构建的树
     *
     * @return 树的根节点
     */
    TreeNode * assignStmt();
```

```
    /**
     * while语句构建的树
     *
     * @return 树的根节点
     */
    TreeNode * whileStmt();

    /**
     * 多项式语句构建的树
     *
     * @return 树的根节点
     */
    TreeNode * expStmt();

    /*
     * 构造简单多项式（无比较符号）语句树
     *
     * @return 树的根节点
     */
    TreeNode * simpleExpStmt(std::list<CompTokenType>::iterator &begin,
                             std::list<CompTokenType>::iterator end);

    /*
     * 构造单项式语句树
     *
     * @return 树的根节点
     */
    TreeNode * termStmt(std::list<CompTokenType>::iterator &begin);

    /*
     * 构造运算单元树
     *
     * @return 树的根节点
     */
    TreeNode * factorStmt(std::list<CompTokenType>::iterator &begin);

    /*
     * 构造语句树时出错的错误处理
     */
    void handleError();

    /**
     * 匹配token
     *
     * @param token 要匹配的token
     */
    void match(TokenType token);

    /*
     * 删除树节点
     *
     * @param root 待删除的树节点的根节点
     */
    void deleteTreeNode(TreeNode* root);

    /* token缓冲区 */
    std::list<CompTokenType> tokenList;

    /* 当前token */
    TokenType token;

    /* 赋值语句等号出现标志，将赋值语句与普通多项式区分开 */
    bool assignStart;

    /* 语法树根节点 */
    TreeNode* treeRoot;
};
```

2.4 代码生成器类 Generator

```
/* 中间代码生成类 */
class Generator {
private:

    /* 用于记录四元式结果项的下标 */
    int resultIndex;

    /* 用于记录四元式结果项 */
    std::string result;

    /* 最后一个四元式标记 */
    std::string lastResult;

    /**
     * 多个语句块
     *
     * @param syntaxTree 语法树
     */
    void mulGen(TreeNode * syntaxTree);

    /**
     * 单个语句
     *
     * @param syntaxTree 语法树
     */
    void sentenceGen(TreeNode * syntaxTree);

    /**
     * while语句
     *
     * @param syntaxTree 语法树
     */
    void whileGen(TreeNode * syntaxTree);

    /**
     * 赋值语句
     *
     * @param syntaxTree 语法树
     */
    void assignGen(TreeNode * syntaxTree);

    /**
     * 表达式语句
     *
     * @param syntaxTree 语法树
     */
    void expGen(TreeNode * syntaxTree);
```

```
    /**
     * 创建一个四元组
     *
     * @param no 编号
     * @param op 运算符号
     * @param arg1 参数1
     * @param arg2 参数2
     * @param result 结果
     * @param backNo 需要回填的编号
     */
    Tuple4 newTuple4(int no, std::string op, std::string arg1, std::string arg2,
        std::string result, int backNo);

    /**
     * 更新结果的数量的下标
     *
     * @param op 操作符
     */
    void updateResultNumber(TokenType op);

    /**
     * 更新结果字符串表示形式
     *
     * @param op 操作符
     */
    void updateResultLabel(TokenType op);

    /**
     * 具有判断性质的语句执行完成之后
     * 更新跳转到下一条语句的标号
     */
    void updateTuple4();
public:

    /* 用于记录四元式列表中的标号 */
    int number;

    /* 四元组列表 */
    std::vector<Tuple4> tuple4List;

    /* 构造函数 */
    Generator();

    /**
     * 生成中间代码
     *
     * @param syntaxTree 语法树
     */
    void codeGen(TreeNode * syntaxTree);

    /**
     * 运行代码生成器
     *
     * @param codeFile 生成目标代码的文件
     */
    static void runGenerator(TreeNode * syntaxTree, std::string codeFile);
};
```

三 主要数据结构设计

```
/* 单词类型 */
typedef enum {
    /*
    * 文件结束
    * 错误的单词: 0x100
    * 注释:       0x101
    * 空格:       0x102
    */
    ENDFILE, TOKEN_ERROR, COMMENT, SPACE,

    /**
    * 标志符: 0x104
    * 布尔型: 0x105
    * 字符型: 0x106
    * 整型:   0x107
    * 浮点型: 0x108
    * 字符串: 0x109
    */
    ID, CONST_BOOL, CONST_CHAR, CONST_INT, CONST_FLOAT, CONST_STR, CONST_INT8,
        CONST_INT16,

    /**
    * 关键字: 0x103
    *
    * abstract, boolean, break, byte, case, catch, char, class, const, continue,
    * default, do, double, else, extends, false, final, finally, float, for, goto,
    * if, implements, import, instanceof, int, interface, long, native, new, null,
    * package, private, protected, public, return, short, static, super, switch,
    * synchronized, this, throw, throws, transient, true, try, void, volatile, while
    */
    ABSTRACT, BOOLEAN, BREAK, BYTE, CASE, CATCH, CHAR, CLASS, CONST, CONTINUE,
    DEFAULT, DO, DOUBLE, ELSE, EXTENDS, JAVA_FALSE, FINAL, FINALLY, FLOAT, FOR, GOTO
        ,
    IF, IMPLEMENTES, IMPORT, INSTANCEOF, INT, INTERFACE, LONG, NATIVE, NEW,
        JAVA_NULL,
    PACKAGE, PRIVATE, PROTECTED, PUBLIC, RETURN, SHORT, STATIC, SUPER, SWITCH,
    SYNCHRONIZED, THIS, THROW, THROWS, TRANSIENT, JAVA_TRUE, TRY, VOID, VOLATILE,
        WHILE,

    /**
    * 赋值运算符号: 0x110
    * = += -= *= /= %= &=
    * ^= |= >>= <<= >>>=
    */
    ASSIGN, ADD_ASSIGN, MINUS_ASSIGN, MUL_ASSIGN, DIV_ASSIGN, MOD_ASSIGN, AND_ASSIGN
        ,
    XOR_ASSIGN, OR_ASSIGN, RIGHT_SHIFT_ASSIGN, LEFT_SHIFT_ASSIGN,
        ZERO_FILL_RIGHT_SHIRT_ASSIGN,
```

```
    /**
    * 关系运算符号
    * ?:   0x111
    * ||:  0x112
    * &&:  0x113
    * | :  0x114
    * ^ :  0x115
    * & :  0x116
    */
    TRIPLE_CMP, OR, AND, OR_BIT, XOR, AND_BIT,

    /**
    * 比较运算符|移位运算符
    * == !=        : 0x117
    * < > <= >=    : 0x118
    * << >> >>>    : 0x119
    */
    EQU, NE, LT, GT, LE, GE, LEFT_SHIFT, RIGHT_SHIFT, ZERO_FILL_RIGHT_SHIRT,

    /**
    * 数值计算符号
    * + -                       : 0x11a
    * * / %                     : 0x11b
    * ++ -- +(正) -(负) ! ~     : 0x11c
    */
    ADD, MINUS, MUL, DIV, MOD, INC, DEC, POSITIVE, NEGATIVE, NOT, NOT_BIT,

    /**
    * 界限符
    * [] () .                   : 0x11d
    * ,                         : 0x120
    * {}                        : 0x121
    * ;                         : 0x122
    */
    BRACKET_ML, BRACKET_MR, BRACKET_SL, BRACKET_SR, BRACKET_LL, BRACKET_LR, DOT,
        COMMA, SEMICOLON

} TokenType;

/* 语法树节点类型 */
typedef enum {
    STMTK, EXPK
} NodeKind;

/* 语句类型 */
typedef enum {
    ASSIGNK, IFK, WHILEK
} StmtKind;

/* 表达式类型 */
typedef enum {
    OPK, NUMK, IDK
} ExpK;
```

```
/* 组合token */
typedef struct {
    TokenType type;
    std::string str;
} CompTokenType;

/* 语法树节点 */
struct TreeNode {
public:
    TreeNode() {
        sibling = NULL;
        child.clear();
    }
    /* 孩子节点 */
    std::vector<TreeNode*> child;
    /* 兄弟节点 */
    TreeNode* sibling;
    /* 所在行号 */
    int lineno;
    /* 节点类型 */
    NodeKind nodeKind;
    /* 语句类型 */
    union {StmtKind stmtKind; ExpK expK;};
    /* 节点属性 */
    union {
        /* 操作符 */
        TokenType op;
        /* 整型常数 */
        int num;
    };
    /* 标志符名称 */
    std::string id;
};

/**
* TraceSource = 1则输出源代码
*/
extern int TraceSource;

/**
 * TraceScan = 1则输出扫描结果
 */
extern int TraceScan;
```

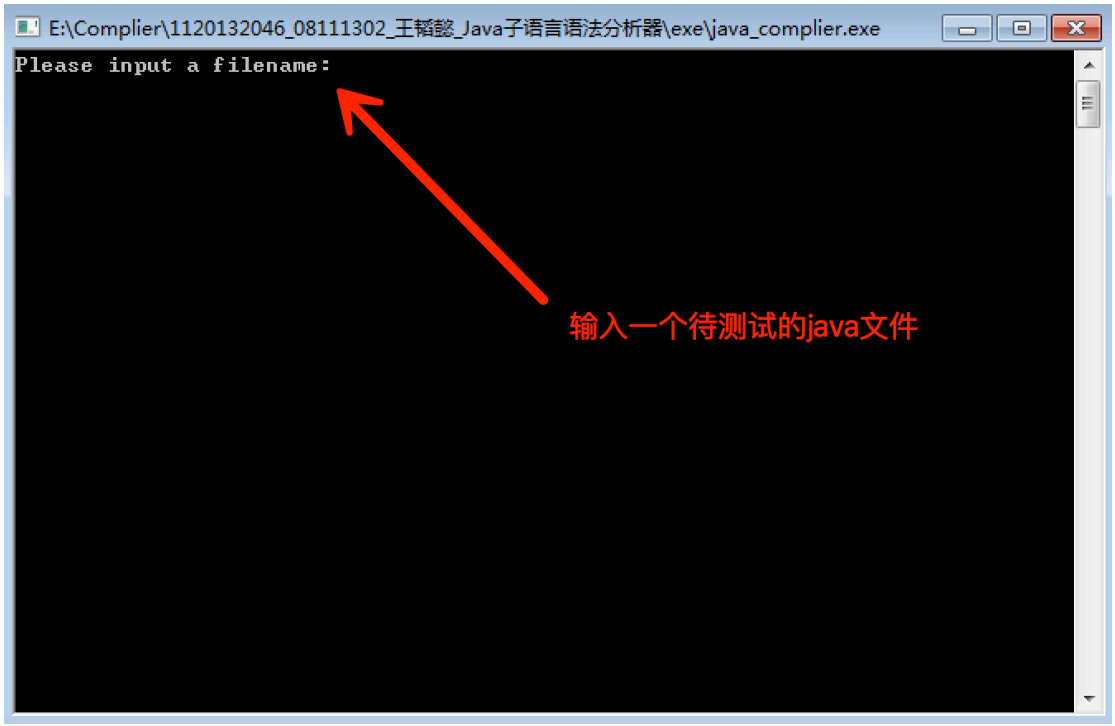
```
/* DFA的状态 */
typedef enum {
    /**
    * 开始，结束，读入标志符，读入字符常量，读入字符串常量，读入整型，读入8进制数，读入16进制数，读入浮点
        型，读入转义符号，读入界限符，读入注释
    * 读入+ - * / % & = < > | ! ^
    * 读入负数
    */
    START, DONE, IN_ID, IN_CONST_CHAR, IN_CONST_STR, IN_INT, IN_INT8, IN_INT16, IN_FLOAT,
        IN_ESC, IN_DELIMETER, IN_COMMENT,
    IN_ADD, IN_MINUS, IN_MUL, IN_DIV, IN_MOD, IN_AND, IN_ASSIGN, IN_LT, IN_GT, IN_OR, IN_NOT
        , IN_XOR, IN_NEGATIVE

} DFAStateType;

/* Token的信息 */
typedef struct {
    /* token所在行号 */
    int lineNumber;
    /* token类型名称 */
    std::string typeName;
    /* token类型 */
    TokenType type;
    /* token的值 */
    std::string value;
    /* token属性字 */
    std::string attr;
} Token;

/* 保存token的错误信息 */
typedef struct {
    /* 错误所在行号 */
    int lineNumber;
    /* 错误的位置 */
    int errorPos;
    /* 该行内容 */
    std::string lineBuf;
    /* 错误的单词 */
    std::string errorToken;
} TokenErrorInfo;

/* 四元式结构 */
typedef struct {
    /* 编号 */
    int no;
    /* 操作符 */
    std::string op;
    /* 参数1 */
    std::string arg1;
    /* 参数2 */
    std::string arg2;
    /* 结果 */
    std::string result;
    /* 以下均为辅助使用 */
    /* 需要回填的编号 */
    int backNo;
} Tuple4;
```

四 运行说明

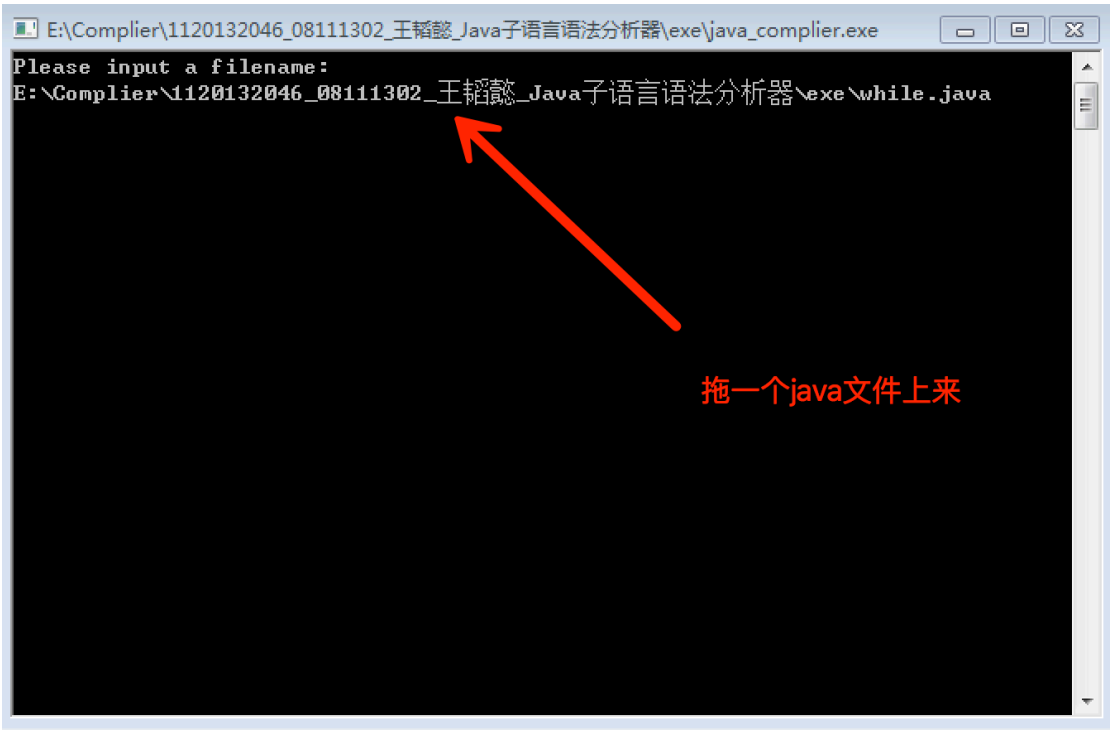
4.1 文件列表

4.2 运行方法

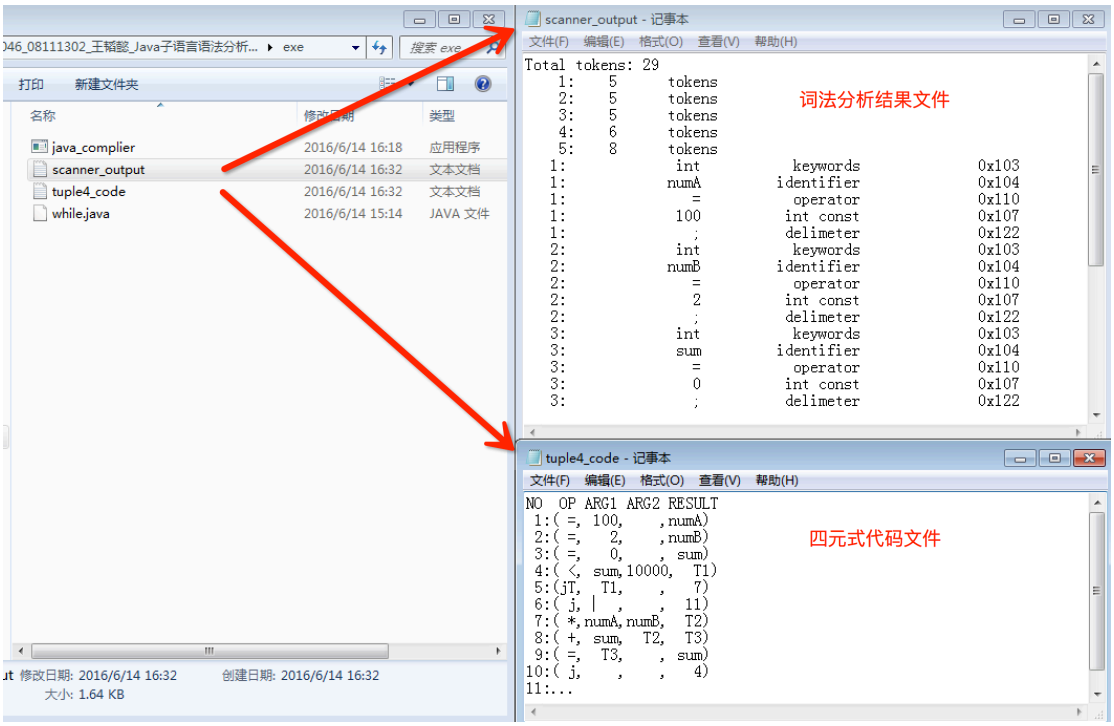
在 exe 文件夹中运行 java_complier.exe 文件得到下面这个界面：



然后拖一个 java 文件上来：



然后回车即可得到输出文件：



五 实现功能

该程序实现了实验要求中的 java 子语言的语法语义以及代码生成功能。