编译实验设计文档（21373450）柳浩东

目录

1. 编译器总体设计

1.1 介绍

在本次实验中，我选择了实现生成目标代码为LLVM编译器，整体采用JAVA语言编写。

1.2 总体结构

本编译器采用的设计为传统的五阶段模式的子集：词法分析，语法分析，错误处理，语义分析和目标代码生成。由于直接生成了LLVM，本编译器没有中间代码生成阶段。

1.3 接口设计

数据流：

testfile.txt→词法分析→语法分析→→→→→语义分析以及目标代码生成

↓ ↓ ↑（无错误，允许生成许可）

错误处理 →→→→→→→

AST:

通过词法分析和语法分析后，生成语法树，错误处理和语义分析接收AST。每个AST的节点Node都含有error_check()和gen_llvm()函数，通过对AST的递归调用进行错误处理，语义分析和代码生成。

下面是具体的调用部分：

```
Lexer lexer = new Lexer();
try (FileInputStream fis = new FileInputStream("testfile.txt");
      PushbackInputStream pbin = new PushbackInputStream(fis)) {
      while (lexer.doSys(pbin) != - 1); //词法分析部分
} catch (IOException ex) {
        ex.printStackTrace();
}
Parser.test();


public static void test() {
        getSym();
        CompUnitNode compUnitNode = CompUnit();//语法分析生成AST
        compUnitNode.ErrorCheck();  //错误处理
        if(Error.count != 0){ //存在错误
            try {
                PrintStream fileOut = new PrintStream("error.txt");
                System.setOut(fileOut);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            }
            ErrorData.sortErr();
            ErrorData.Print(); //输出错误
        } else { //无错误，进行代码生成
            isErrorCheck = false;
            try {
                PrintStream fileOut = new PrintStream("llvm_ir.txt");
                System.setOut(fileOut);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            }
            resetSym();
            SymbolManager.resetSym();
            getSym();
            CompUnitNode compUnitNode1 = CompUnit();
            compUnitNode1.genIR(); //代码生成
        }
    }
```

1.4 文件组织

主要分为4个文件夹：

TokenAnalyse:

词法分析部分，负责根据输入的testfile.txt生成token流。里面用Hashmap存储了所有关键词和操作符。

SyntaxAnalyse:

负责根据词法分析得到的token流生成AST树，主要是各种节点，如decl_node,ident_node,lval_node等。和分析token流的parser模块。

ErrorCheck:

主要是符号表，错误的种类，具体错误的信息（位置，类型），单例模式的符号表管理员。

LLVM：

负责从语法树生成LLVM代码，主要包括LLVM的经典架构：module，user,value,basic_block等。还有LLVM的具体指令类型，具体的每一条LLVM指令，单例模式的llvm_builder。类型系统

Complier.java：编译器的入口，依次调用词法分析，语法分析，错误处理，代码生成。

2. 词法分析设计

2.1 编码前的设计

读取输入流，拼接成一个个的字符串对象。

我将Token分为：

1。标识符

2。关键词

3。单字符操作符

4。双字符操作符

5。数字

6。FormatString

判断的方法如下：

如果首字符为字符或下划线，则读取到空字符串为止，将得到的字符串首先查关键词表，查到则为对应关键词，查不到则为ident。

如果首字符为数字，则读取到空字符串为止，计算出得到的数字。

如果首字符为引号，则解析出FormatString。

剩下的情况只能是单字符操作符或双字符操作符，首先预读一个字符，尝试匹配双字符操作符，匹配失败则回退，匹配单字符操作符。

下面是具体的关键字和操作符表：

单词名称	类别码	单词名称	类别码	单词名称	类别码	单词名称	类别码
Ident	IDENFR	!	NOT	*	MULT	=	ASSIGN
IntConst	INTCON	&&	AND	/	DIV	;	SEMICN
FormatString	STRCON	||	OR	%	MOD	,	COMMA
main	MAINTK	for	FORTK	<	LSS	(	LPARENT
const	CONSTTK	getint	GETINTTK	<=	LEQ	)	RPARENT
int	INTTK	printf	PRINTFTK	>	GRE	[	LBRACK

break	BREAKTK	return	RETURNTK	>=	GEQ	]	RBRACK
continue	CONTINUETK	+	PLUS	==	EQL	{	LBRACE
if	IFTK	-	MINU	!=	NEQ	}	RBRACE
else	ELSETK	void	VOIDTK				

下面是Token的部分设计：

```
public class Token {
    private TokenType type;
    private String value;
    private int lineNum;
    public Token(TokenType type, String value, int lineNum) {
        this.type = type;
        this.value = value;
        this.lineNum = lineNum;
    }
}
```

2.2 编码完成之后的修改

在处理注释的时候，需要考虑避免和除法符号"/"的混淆。处理注释的时候仍然需要记录换行符的存在，而不是全部忽略，不然后面的行号都会出错。

原本提前看一个符号采用的思路是先取出整个字符串，通过改变当前指针的位置来实现，后来发现Java本身提供了PushbackInputStream输入流类。

主要的构造函数是：

```
public PushbackInputStream(InputStream in, int size)
```

以下是一些常用的方法：

```
void unread(int b)//将指定字节推回到流中。如果推回多个字节，可以使用 void unread(byte[]
b, int off, int len) 方法。

int read()//从输入流读取一个字节。

int read(byte[] b, int off, int len)//从输入流读取字节数组。

void close()//关闭输入流。
```

3. 语法分析设计

3.1 编码前的设计

采用**递归下降**的思路，整体上看大多数为LL(1)分析，有少部分为LL(N)分析。

大部分的文法可以根据第一个字符决定选择的推导序列，有的非终结符只有一个推导，这些都能用LL(1)实现。

首先**消除左递归**：对所有的形如

```
S->SA
```

改写为

```
S->AA*
```

具体有：

```
AddExp -> MulExp {('+' | '-') MulExp}
EqExp -> RelExp {('==' | '!=') RelExp}
RelExp -> AddExp {('<' | '>' | '<=' | '>=') AddExp}
......
```

部分地方的LL(N)：

```
LVal '=' Exp ';'

LVal '=' 'getint''('')'';'
```

这样的地方需要先调用parseLVal()函数对若干个终结符进行解析，然后再解析“=”，然后判断是不是getInt对应的token。

回溯问题：如上面的解析，在解析后需要回溯到LVal之前，我的策略是在可能需要回溯的地方先预先存储下当前解析到的位置，根据该存储位置回溯。

具体代码如下：

```
            int curPos = TokenStream.getPos(); //存储了当前的位置以便回溯
            LValNode lValNode = lVal();
            if(curToken.getType() == TokenType.ASSIGN){
                getSym();
                if(curToken.getType() == TokenType.GETINTTK) {
                    TokenStream.setPos(curPos-1); //回溯
                    getSym();
                    GetIntStmtNode getIntStmtNode = getIntStmt();
                    return new StmtNode(line,getIntStmtNode,9);
                } else {
                    TokenStream.setPos(curPos-1);
                    getSym();
                    AssignStmtNode assignStmtNode = assignStmt();
                    return new StmtNode(line,assignStmtNode,1);
                }
            } else {
                TokenStream.setPos(curPos-1);
                getSym();
```

```
                ExpStmtNode expStmtNode = expStmt();
                return new StmtNode(line,expStmtNode,2);
            }
```

3.2 编码完成之后的修改

事实上，语法分析不一定要建立语法树。我一开始也打算一边解析一边输出语法成分，但是考虑到后面的错误处理和语义分析，我还是建立了语法树。

基本上大部分的语法成分都有对应的节点，如break_node,return_node,add_node等。

还有非终结符对应的节点：func_node,func_param_node,var_decl_node等。

节点的父类的设计：

```
public class Node {
    protected int beginLine;
    protected SyntaxType syntaxType;

    public Node(int beginLine) {
        this.beginLine = beginLine;
    }
    public void PR(){

    }
    public void ErrorCheck(){

    }
    public Value genIR(){
        return null;
    }
    public int eval(){
        return -114514;
    }
    protected BasicBlock continueBranch = null;
    protected BasicBlock breakBranch = null;
    public void setContinueBranch(BasicBlock continueBranch) {
        this.continueBranch = continueBranch;
    }
    public void setBreakBranch(BasicBlock breakBranch) {
        this.breakBranch = breakBranch;
    }
}
```

```
//程序入口
public static CompUnitNode CompUnit(){//整个C程序的节点
    ArrayList<DeclNode> declNodes = ZeroOrMoreDecl();//0个或多个变量声明节点
    ArrayList<FuncDefNode> funcDefNodes = ZeroOrMoreFuncDef();//0个或多个函数声明节
点
    MainFuncDefNode mainFuncDefNode = mainFuncDef();//主函数声明节点
    CompUnitNode compUnitNode = new
        CompUnitNode(curLine,declNodes,funcDefNodes,mainFuncDefNode);
    return compUnitNode;
}
```

4. 错误处理设计

4.1 编码前的设计

在语法分析建立的AST树上，采用树的后序遍历方法，依次调用各个叶子节点和根节点的错误处理程序。

题目中明确表示不会出错的地方，节点的ErrorCheck为空函数。题目中表示可能出错的地方，按照可能出错的类型设计错误处理程序。

错误处理的过程中，为了识别符号未定义，重定义等错误。建立各个层次的符号表是必要的，每次进入一个AST的block_node或func_decl节点。就创建一个新的符号表，各个符号表之间有树的根节点和叶子节点的关系，在祖先节点符号表中定义的变量和函数，在后代节点符号表中可以被正确识别。

符号表中函数的表示：

```
public class FuncSymbol {
    private int line;

    private ArrayList<Param> params;

    private String name;

    private int type; // 1 void 2 int

    private int elementNum;

    private ArrayList<Integer> degrees; // 0 int 1 int[] 2 int[][i]

    public FuncSymbol(int line, String name, int type, int elementNum,
ArrayList<Integer> degrees) {
        this.line = line;
        this.name = name;
        this.type = type;
        this.elementNum = elementNum;
        this.degrees = degrees;
    }
}
```

符号表中变量的表示：

```
public VarSymbol(String name, int line, boolean isGlobal, ArrayList<Integer>
dims, ArrayList<Integer> initVals,Type type,String llvmName) {
    this.name = name;
    this.line = line;
    this.isGlobal = isGlobal;
    this.dims = dims; //各维度长度
    this.degree = dims.size();//维数
    this.values = initVals;
    this.type = type;
    this.llvmName = llvmName;
}
```

对于缺少分号，缺少左括号这样的语法错误，则在词法分析的时候检查，即在对应地方检查是否匹配到了正确的token符

具体举例：

```
if(curToken.getType()!=TokenType.SEMICN) {
    ErrorData.AddError(new Error(lastLine,'i'));
} else {
    getSym();
}
```

4.2 编码完成之后的修改

在实际的错误处理中，函数参数类型不匹配不需要检查具体的维度值，只需要检查维度的个数是否相同即可。

在处理在非循环块中使用continue和break语句的时候，需要在符号表中设置一个全局静态变量，标志当前是否在循环块中。

由于规定同一行最多只有一个错误，在函数参数检查的时候，需要首先检查是否声明了该变量，在声明没有出错的情况下，再检查该变量的类型是否匹配，如果变量未定义，则不再进行维度的检查。

5. 代码生成设计

5.1 编码前的设计

代码生成的思路类似于错误处理，在每一个节点都设置一个gen_llvm函数，从根节点开始调用它们。

根节点:

```
public Value genIR(){
    ArrayList<GlobalVars> globalVarsList = new ArrayList<>();
    ArrayList<FunctionDecl> functionDecls = new ArrayList<>();
    for(DeclNode declNode:declNodes) {
        GlobalVars globalVars = (GlobalVars) declNode.genIR(); //生成全局变量LLVM
        globalVarsList.add(globalVars);
    }
    for (FuncDefNode funcDefNode:funcDefNodes) {
        FunctionDecl functionDecl = (FunctionDecl) funcDefNode.genIR();//生成函数声明LLVM
        functionDecls.add(functionDecl);
    }
    FunctionDecl mainFunction = (FunctionDecl) mainFuncDefNode.genIR();//生成Main函数LLVM
    Module module = new Module(globalVarsList,functionDecls,mainFunction);//构建LLVM的module对象
    module.print();//输出LLVM
    return Nothing;
}
```

LLVM的函数，变量的结构和文法中定义的类似

```
public FunctionDecl(String name, ArrayList<Param> paramList,
ArrayList<BasicBlock> BBList, Type retType) {
    super(name);
    this.paramList = paramList;
    this.BBList = BBList;
    this.retType = retType;
    IRbuilder.setLocalVarCntMap(this);
}
```

```
public GlobalVar(String name, boolean isConst,
ArrayList<Integer>dims,ArrayList<Integer> initVal,Type type) {
    super(name);
    this.isConst = isConst;
    this.toPrint = isConst?"constant":"global";
    this.initVals = initVal;
    this.dims = dims;
    this.type = type;
    this.isArr = dims.size() > 0?true:false;
}
```

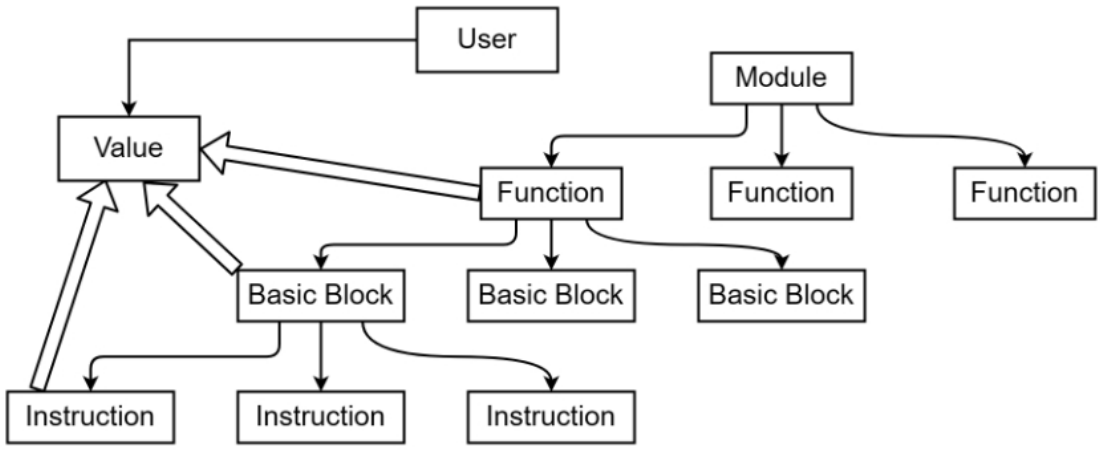
对于声明语句声明的变量以及函数的形式参数，使用llvm的alloc语句为其分配空间再使用。计算过程中的临时变量则直接使用，这里为每个函数维护一个变量计数器，用于分配新变量时的命名。

具体翻译举例:

```
//这是把addExp翻译成llvm的函数
public Value genIR(){
        if(type == 1) {
            return mulExpNode.genIR();
        } else {
            Value op1 = addExpNode.genIR(); //返回存储了addExpNode.genIR()计算结果的
变量
            Value op2 = mulExpNode.genIR(); //返回存储了mulExpNode.genIR()计算结果的
变量
            IRsym distinct = IRbuilder.allocNewSym(((IRsym)
op1).getVarBaseType());//申请一个新变量
            if (opt == 1) {
                //新变量=变量1+变量2
                Instr instr = new Instr(InstrType.add, distinct, (IRsym) op1,
(IRsym) op2);
                instr.setFullType(((IRsym) op1).getVarBaseType());
                IRbuilder.addInstrToBB(instr);//IRbuilder为单例模式，负责分配各个变
量，把llvm指令加入对应基本块，申请新基本块等工作，所有的instr的构建都基于它
            } else {
                Instr instr = new Instr(InstrType.sub, distinct, (IRsym) op1,
(IRsym) op2);
                instr.setFullType(((IRsym) op1).getVarBaseType());
                IRbuilder.addInstrToBB(instr);
            }
            return distinct;
        }
    }
```

Value和User

所有的gen_llvm的返回值均为Value



理论上，一切LLVM中的类都可以是Value的子类，但我的编译器中，Value的主要作用是返回中间计算过程的临时变量名。如a=b+c+d;它的llvm如下

```
%3 = add i32 %1, %2 //计算b+c
%4 = load i32, i32* @d
%5 = add i32 %3, %4 //计算(b+c)+d
store i32 %5, i32* @a//存入a
```

完成了b+c的计算后，需要一个临时变量存储它的值，并且该变量还需要能够被后续的计算过程访问到，在此我借助了Value作为返回值，让后续的计算能够以之前的结果作为变量。

User类：一切能够使用其他Value的类均为User类

5.2 编码完成之后的修改

在生成过程中，我遇到的最大的一个问题就是：LLVM代码不允许有空的基本块以及LLVM代码的基本块只能以br或者ret语句结尾，而且br或ret不能有多条。

下面给出几种非法基本块

```
//错误1
%3 = add i32 %1, %2
ret %5
%3 = add i32 %1, %2 //br或者ret后不能有新语句
```

```
//错误2
%3 = add i32 %1, %2
//必须有br或ret
```

```
//错误3
%3 = add i32 %1, %2
ret %5
ret %5 //最多有一条br或ret
```

虽然正常的程序员不会在break语句后面写新的语句，但是测试样例仍旧考虑了这种情况，导致了一开始设计的结果发生了错误1这样的情况，对此我的解决办法是在构建语法树的时候，就把break后面的部分直接舍弃。但这样又导致了我错误处理的遗漏，后来给出的折中办法是：对错误处理和代码生成分别构建一次语法树。

错误2主要发生在基本块的直接进入上，及时基本块3后面按照顺序理应为基本块4，仍然需要在基本块3的结尾加上br语句来进入4，这里的解决办法为检查那些没有以br或ret结尾的基本块，给它们加上直接后继基本块的br语句。

错误3主要发生在嵌套的if语句的上，当一个基本块既是内部if的结尾，又是外部if的结尾，它的末尾就会被添加上两条br语句，对此我的解决办法是在添加br语句之前检查该基本块是否以及具有br语句，如果有了则不再添加。

6. 总结

虽然这次我的编译器选择了生成LLVM的路线，相比那些生成MIPS再努力优化的同学要容易许多，但依然也给我带来了不小的挑战。通过这一个学期的学习，我基本上理解了高级语言程序是如何在计算机上真正运行的，更加重要的是，这次的编译课程让我学会了量力而行。面对期末繁重的课程复习压力，即使现在离竞速排序结束还有许多的时间，我还是决定到此为止了。这是我上大学以来第一次主动放弃，在此也祝那些选择了竞速排序的同学拿到与自身努力相匹配的成绩。回到我自己的编译器，从一开始的文法分析自己编写样例，到词法分析，语义分析生成LLVM。编译器的模型在我脑中一点一点变得清晰，让我意识到了编程没有捷径可走，看起来功能再强大的程序，也是程序员一条一条语句，一点一点写出来的。并非我之前想的小程序用小积木拼接，大程序用大积木拼接。在LLVM建类初期的迷茫也让我意识到了好的架构远能力比编程能力更加难得。坦白来说，我放弃生成MIPS的一部分原因也是我的架构中存在的太多耦合杂糅。它们的存在使我不敢拿期末考试冒险。越大的程序，越不能着急，好的架构在编写的过程中可以带来快乐，仿佛编译器本身就存在与此，我们像考古一样一点一点挖掘出来；不好的架构只能带来无尽的烦恼和额外的条件判断。

最后也是为后来的同学提出一些建议：MIPS固然强大，但并不一定合适。生成LLVM也是一条不错的选择。