《编译技术》课程设计文档

学号：21371245 姓名：孙玉粮

一、参考编译器介绍

我参考的编译器是实验网站附件中的PL/0简单编译系统的结构。PL/0为编译——解释执行程序。编译部分生成的目标代码为PCODE指令。我在前期选择参考编译器时主要参考的是前端（词法分析、语法分析、语义分析生成中间代码）的架构。PL/0采用一遍扫描，以语法分析为核心，在语法分析过程中调用词法分析程序取单词，同时也进行语义分析，最终生成目标代码。同时，它进行语义检查，发现错误后就转出错处理程序。

词法分析中，`getsym`子程序用于跳过所有空格读取单词，并将单词符号放入`wsym`或`ssym`中；

语法分析为核心的主程序，PL/0在语法分析过程中调用其他子程序，语法分析中有多个递归子程序；

出错处理中，`error(n : integer )`用于输出错误信息；

符号表管理中，使用全局变量`table`作为符号表,定义如下:

```
      table : array[0..txmax] of
                record
                  name : alfa;
                  case kind: objecttyp of
                    constant : (val:integer );
                    variable,prosedure: (level,adr: integer )
                end;
```

二、编译器总体设计

总体结构与文件组织：

编译器的总体结构与文件组织如下所示。

```
└──src
    |   Compiler.java
    |   config.json
    |   IRPort.java
    |   IRVisitor.java
    |   IRVisitor1.java
    |   Main.java
    |   src.zip
    |   ValueTable.java
    |   Visitor.java
    |
    ├──EntryType
    |       Array1.java
    |       Array2.java
    |       ConstArray1.java
    |       ConstArray2.java
    |       ConstValue.java
```

```
|       ConstVar.java
|       FuncParam.java
|       FuncType.java
|       Var.java
|
├──Instructions
|       BinaryOperator.java
|       BinaryOpType.java
|       Instruction.java
|       MemoryInstr.java
|       MemoryType.java
|       TerminatorInstr.java
|       TerminatorType.java
|
├──Lexer
|       Lexer.java
|       Symbol.java
|       Token.java
|
├──Parser
|       ASTNode.java
|       ErrorNode.java
|       ErrorType.java
|       GrammerSymbol.java
|       Parser.java
|
├──STable
|       ReferencedEntry.java
|       SymbolTable.java
|       TableEntry.java
|       TableEntryType.java
|
├──Type
|       ArrayType.java
|       FunctionType.java
|       IntegerType.java
|       LabelType.java
|       PointerType.java
|       Type.java
|       VoidType.java
|
└──Values
    |   Argument.java
    |   BasicBlock.java
    |   Module.java
    |   User.java
    |   Value.java
    |
    └──Constants
            Constant.java
            ConstantArray.java
            ConstantInt.java
            Function.java
            FunctionInt.java
            FunctionVoid.java
            GlobalVariable.java
```

接口设计

我的接口主要都在主文件`Compiler.java`中，其中`initMap`静态类初始化错误处理中的错误对照表，输出方法封装在类中，内容如下：

```
public class Compiler {
    private static final HashMap<ErrorType, String> errorType2Symbol = new
HashMap<>();

    private static void initMap() {
        errorType2Symbol.put(ErrorType.IllegalChar, "a");
        errorType2Symbol.put(ErrorType.IdentRedefined, "b");
        errorType2Symbol.put(ErrorType.IdentUndefined, "c");
        errorType2Symbol.put(ErrorType.ParaNumNotMatch, "d");
        errorType2Symbol.put(ErrorType.ParaTypeNotMatch, "e");
        errorType2Symbol.put(ErrorType.ReturnTypeError, "f");
        errorType2Symbol.put(ErrorType.ReturnMissing, "g");
        errorType2Symbol.put(ErrorType.ConstAssign, "h");
        errorType2Symbol.put(ErrorType.SEMICNMissing, "i"); // ; missing
        errorType2Symbol.put(ErrorType.RPARENTMissing, "j"); // ) missing
        errorType2Symbol.put(ErrorType.RBRACKMissing, "k"); // ] missing
        errorType2Symbol.put(ErrorType.PrintfFormatStrNumNotMatch, "l");
        errorType2Symbol.put(ErrorType.BreakContinueNotInLoop, "m");
    }

    public static void main(String[] args) {
        try (
                BufferedReader reader = new BufferedReader(new
FileReader("testfile.txt"));
                BufferedWriter output = new BufferedWriter(new
FileWriter("output.txt"));
                BufferedWriter output1 = new BufferedWriter(new
FileWriter("output_word.txt"));
                BufferedWriter output2 = new BufferedWriter(new
FileWriter("error.txt"));
                BufferedWriter output3 = new BufferedWriter(new
FileWriter("llvm_ir.txt"));
                BufferedWriter output4 = new BufferedWriter(new
FileWriter("mips.txt"));
                // BufferedWriter  output2 = new BufferedWriter(new
FileWriter("output_image.txt"))
        ) {
            Lexer lexer = new Lexer(reader, output1);
            lexer.parse();
            Parser parser = new Parser(lexer.getAllTokens(), false);
            ASTNode root = parser.parse();
            // output the AST to output.txt
            parser.printAST(root, output);
            // parser.printASTImage(root, output2);

            // System.out.println("Parser finished.");
            // mistake process

            // check printf("Hello world$"); (printf format string error) l
            Compiler.initMap();
            Visitor visitor = new Visitor(root, false, errorType2Symbol);
            visitor.visitCompUnit(root);
```

```
            visitor.printToFile(output2);

            if (visitor.getErrorListSize() > 0) {
                System.out.println("has error!");
                return;
            }

            // Middle Code Generation LLVM IR
            Module module = new Module();
            IRVisitor1 irVisitor = new IRVisitor1(root, false, module);
            // IRVisitor irVisitor = new IRVisitor(root, true, module);
            irVisitor.visitCompUnit(root);
            output3.write(module.toString());
            output3.flush();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

三、词法分析设计

编码前的设计

`Lexer`文件夹下`Lexer.java/Symbol.java/Token.java`

- 词法分析器，用于读入源程序，分析单词，输出单词类别码
- `Symbol.java`是枚举类，用于存储所有的类别码
- `Lexer.java`中的方法：
  - `parse()`：执行词法分析
  - `nextSymbol()`：读入下一个单词
  - `print()`：按照作业要求格式输出读到的单词和类别码。

编码完成之后的修改

- 在后续语法分析过程中为解析方便新增`Token.java`用于存储单词，每个`Token`包含：
  - 名称`value`
  - 对应的类别码`symbol`
  - 所在行号`lineNumber`
  - 在行中的索引`index`
- 在每次读完一个单词后，缓存名称的`StringBuilder token`忘记清空
- 对于`/`、`/*`和`//`需要进一步预读来判断。
- 在预读单词时需要判断当前字母位置是否超过行的最长字符数，否则会报错。如以下判断`=`和`==`的情况：
  - ```
    int a =
        1;
    ```

四、语法分析设计
    ```

编码前的设计

构造递归下降程序，对程序的词法分析结果进行递归下降分析，最后获取语法树。

`Parser`文件夹下`ASTNode.java/GrammerSymbol.java/Parser.java`

- `GrammerSymbol.java`是枚举类，用于存储语法成分
- `ASTNode.java`是抽象语法树的节点类，每个节点存储：
  - `private GrammerSymbol grammerSymbol;`：对应的语法成分（非终结符）
  - `private Token token;`：终结符对应的`token`
  - `private ArrayList<ASTNode> children;`：多叉树的孩子节点
  - `private ASTNode parent;`：该节点的父节点
  - `private boolean isLeaf;`：该节点是否为多叉树的叶节点
- `Parser.java`是执行语法分析递归下降过程的类。

编码完成之后的修改

- 第一次写的时候对于几个左递归文法没有按照常规的语法树构建去做。比如对`AddExp -> MulExp | AddExp ('+' | '-') MulExp`，一开始我的语法树是BNF形式的，即`AddExp -> MulExp{('+' | '-') MulExp}`，在这种情况下，我加了一个`addSelfNode`方法来解决输出上的问题，但是没有实质解决语法树的构建问题。在后来的错误处理语义分析时再递归寻找错误时发现这种错误的语法树会带来不小麻烦，于是采用将左侧成分向上打包一层的方法构建正确的语法树。

  ```
  while (curToken().getSymbol() == Symbol.PLUS
                  || curToken().getSymbol() == Symbol.MINU) {
              // addSelfNode(addExp, depth);
              ASTNode temp = new ASTNode(GrammerSymbol.AddExp, null,
  depth);
              temp.addChild(addExp);
              addExp.setParent(temp);
              addExp.setDepth(depth + 1);
              child.setDepth(depth + 2);
              addExp = temp;

              addCurToken(addExp, depth);

              child = parseMulExp(depth + 1);
              child.setParent(addExp);
              addExp.addChild(child);
          }
  ```

五、错误处理设计

编码前的设计

`Parser`文件夹下的`Parser.java/ErrorNode.java/ErrorType.java`，`EntryType`文件夹下的所有符号表项类，
`ReferencedEntry.java/SymbolTable.java/TableEntry.java/TableEntryType.java`以及`Visitor.java`

错误处理大体可以分为两类错误，第一类是`a,i,j,k,l`五类在语法分析阶段就可以分析出的错误，对于这类错误的处理方式是补充一个"错误节点"，目的是使程序能够正常运行到语义分析，错误处理输出的阶段。比如在`ConstDecl -> 'const' BType ConstDef { ',' ConstDef } ';'`中可能会出现`i`类缺少分号的错误，我的处理方式如下：

```
// ';'
if (!Objects.equals(curToken().getValue(), ";")) { // error: i =>
Parser.ErrorType: SEMICNMissing
    int lineNum = allTokens.get(tokenPos - 1).getLineNumber();
    constDecl.addChild(new ErrorNode(ErrorType.SEMICNMissing,
                                     lineNum, constDecl, depth + 1));
    // nextToken();
    // throw new IllegalArgumentException("The end of ConstDecl is not \";\"");
} else {
    addCurToken(constDecl, depth);
}
```

- `Visitor.java`
  - 解决语法分析识别不了的错误。
  - 方法：递归下降式的`visitXXX()`，找出每个语法成分的错误
- `ReferencedEntry.java`
  - 引用参数类，记录对应实际的参数，比如实际变量是一个数组`a[10]`，传入函数的参数是`a[0]`，那么就可以通过该参数类寻找到实际传入函数的值。
- `TableEntryType.java`
  - 枚举类，枚举所有符号表项的种类。
  - ```
    public enum TableEntryType {
        Var, // 变量
        ConstVar, // 常量

        Array1, // 一维数组
        ConstArray1, // 一维常量数组
        Array2, // 二维数组
        ConstArray2, // 二维常量数组

        FunctionVoid, // 无返回值函数
        FunctionInt, // 有返回值函数

        ReferencedEntry // 引用表项,
    }
    ```
- `EntryType`文件夹
  - 该文件夹下为所有符号表项的种类建立对应类
  - 新增`FunctionParam`类专门来记录函数参数，包含子类`Array1, Array2, Var`
- `SymbolTable.java`
  - 符号表类，属性：
    - `private boolean isRoot;`是否为全局符号表
    - `private SymbolTable parent;`栈式符号表的父亲符号表
    - `private ArrayList<SymbolTable> children;`栈式符号表的儿子符号表
    - `private HashMap<String, TableEntry> entries;`符号表项
- `TableEntry.java`
    ```

- 符号表项类，有和所有符号表项种类对应的构造函数用于构造相应的符号表项。

编码完成之后的修改

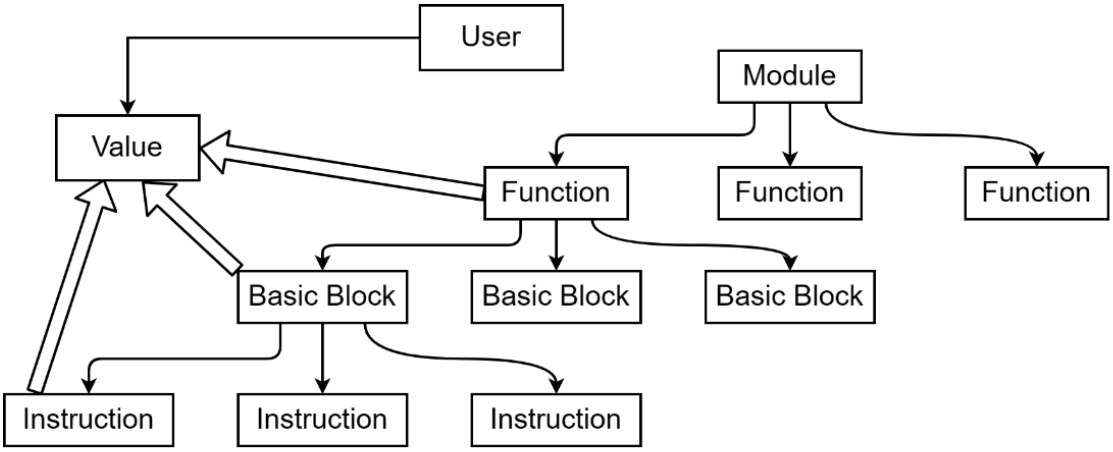
在解析`Exp`等语法成分时，使用`private int curInt;`全局变量记录最终得到的数值，这样保证了递归的持续进行。并添加`private boolean isConstant;` `private boolean isMultiArrayInit;`等标志量保证递归解析数组或常数时的正确性。

六、代码生成设计

编码前的设计

`Values`文件夹，`Type`文件夹，`Instructions`文件夹，`IRPort.java/IRVisitor1.java/ValueTable.java`。

我只按照实验教程进行了`LLVM IR`的生成，没有进行`MIPS`代码的生成。实现的总体思想是“一切皆Value”，这也与LLVM的官方文档思想一致。由于错误处理时的符号表对表项的分类过于繁杂，我在这里新建了一个`ValueTable.java`进行符号表项的管理。同时仍然采用递归下降的方法进行LLVM树的构建。



- `Module.java`
  - 包含所有的`GlobalVariable`和所有的`Function`
  - ```
    public class Module {
        // 顶层模块
        private ArrayList<Function> functions;
        private ArrayList<GlobalVariable> globalVariables;

        public Module () {}
        public void addFunction(Function function) {}
        public Function getFunction(String name) {}
        public void addGlobalVariable(GlobalVariable globalVariable) {}
        public String toString() {}
    }
    ```
- `Value.java`
  - 记录LLVM中所有的内容信息
    ```

-
```
public class Value {
    private static int idCounter;
    private int id;
    protected String name;
    protected Type type;
    protected Value parent;
    protected ArrayList<User> users;

    public Value(String name, Type type, Value parent) {}
    public void addUser(User user) {}
    public Type getType() {}
    public String getName() {}
}
```
-

- `User.java`
  - 存储使用的Value
  -
```
public class User extends Value{
    private ArrayList<Value> values;

    public User(String name, Type type, Value parent, Value... values) {
        super(name, type, parent);
    }
    public Value getUsedValue(int index) {}
    public int getUsedValuesNum() {}
}
```
  -

- `Constant.java`
  - 记录所有的常量，包括常数，常数组，函数，全局变量
  -
```
public class Constant extends User {
    public Constant(String name, Type type, Value parent, Value...
values) {
        super(name, type, parent, values);
    }

    public static Value getZeroConstant(Type type) {}
}
```

- `Function.java`
  - 复用错误处理时的`FunctionInt`和`FunctionVoid`，记录所用的`BasicBlock`和函数类型
  -
```
public class Function extends Constant{
    private FunctionInt functionInt;
    private FunctionVoid functionVoid;
    private int functionType; // 0 for int, 1 for void
    public ArrayList<Argument> getArguments() {}
    public ArrayList<BasicBlock> getBasicBlocks() {}

    public Function(String name, Type type, boolean isBuiltIn, Type...
paramType) {}
```

```
    public Function(String name, boolean isBuiltIn, FunctionInt
functionInt) {}
    public Function(String name, boolean isBuiltIn, FunctionVoid
functionVoid) {}
    public int getFunctionType() {}
    public String getName() {}
    public void addBasicBlock(BasicBlock basicBlock) {}
    public int getParamsNum() {}
    public int getArgsNum() {}
    public String toString() {}
    public ArrayList<Type> getParamsType () {}
}
```

- `BasicBlock.java`
  - 记录含有的`Instructions`
  - ```
    package Values;
    public class BasicBlock extends Value {
        // 基本块中含有指令的列表
        private ArrayList<Instruction> instructions = new ArrayList<>();

        public BasicBlock(String name, Value parent) {}

        public void addInstruction(Instruction ans) {}

        public String toString() {}

        public ArrayList<Instruction> getInstructions() {}
    }
    ```

- `Instruction.java`
  - 记录指令
  - ```
    public class Instruction extends User {

        public Instruction(String name, Type type, Value parent, Value...
    values) {
            super(name, type, parent, values);
        }
    }
    ```

编码后的修改

一开始我的架构不是很清晰，符号表沿用了错误处理时的符号表，导致代码耦合度很高，在后续的Debug以及分析思路时都很困难。后来在渐渐清晰架构的情况下重新建立了符号表，并使用Java中的泛型思想，使得代码耦合度降低，架构也清晰了很多。