编译器设计文档

参考编译器介绍

编译器示例代码-sysy-compiler

总体结构

整体分为前端（词法、语法、AST）、中端（中间代码表示 IR、符号表）、后端（目标代码生成三部分，另外把错误处理与工具模块拆分出来，主入口为 `Compiler.java`。

接口设计

前端

```java
public class FrontEnd {
    private static Lexer lexer;
    private static Parser parser;

    public static void SetInput() throws IOException {
        lexer = new Lexer(IOhandler.GetInput());
        parser = new Parser();
    }

    //词法分析器生成Token流
    public static void GenerateTokenList() throws IOException {
        lexer.GenerateTokenList();
    }

    //语法分析器生成语法树
    public static void GenerateAstTree() {
        parser.SetTokenStream(GetTokenStream());
        parser.GenerateAstTree();
    }

    //取Tokens
    public static ArrayList<Token> GetTokenList() {
        return lexer.GetTokenList();
    }

    //获得Token流
    private static TokenStream GetTokenStream() {
        return new TokenStream(lexer.GetTokenList());
    }

    //取语法树
    public static CompUnit GetAstTree() {
        return parser.GetAstTree();
    }
}

```

中端

```
public class MidEnd {
    private static CompUnit rootNode;
    private static IrModule irModule;

    //创建符号表
    public static void GenerateSymbolTable() {
        SymbolManger.Init();
        rootNode = FrontEnd.GetAstTree();
        rootNode.Visit();
        SymbolManger.GoBackToRootSymbolTable();
    }

    //创建中间代码
    public static void GenerateIr() {
        irModule = new IrModule();
        IrBuilder.SetCurrentModule(irModule);
        Visitor visitor = new Visitor(rootNode);
        visitor.Visit();
        IrBuilder.Check();
    }

    //获得符号表
    public static SymbolTable GetSymbolTable() {
        return SymbolManger.GetSymbolTable();
    }

    //获得中间表示
    public static IrModule GetIrModule() {
        return irModule;
    }
}

```

后端

```
public class BackEnd {
    private static IrModule midEndModule;
    private static MipsModule backEndModule;

    //生成Mips代码
    public static void GenerateMips() {
        backEndModule = new MipsModule();
        MipsBuilder.SetBackEndModule(backEndModule);

        midEndModule = MidEnd.GetIrModule();
        midEndModule.toMips();
        // 进行窥孔优化
        if (Setting.FINE_TUNING) {
            PeepHole peepHole = new PeepHole();
            peepHole.Peep();
        }
    }

```

```
18
19      //获得Mips表示
20      public static MipsModule GetMipsModule() {
21          return backEndModule;
22      }
23  }
24
```

详细分析

前端

- `lexer`：词法分析，负责生成`token`流。
- `parser`：语法分析将`tokens`转为语法树。
- `ast`：完整的抽象语法树节点（`CompUnit`、`FuncDef`、`Stmt`、`Exp`、`Decl`等），对应`parser`所使用的类。
- `FrontEnd.java`：前端的统一接口，输出`AST`、`Tokens`等供后续阶段使用。

中端

- `symbol`
  - `SymbolTable` / `SymbolManager` / 各类`Symbol`：追踪作用域、类型与符号信息，建立符号表
- `visit`
  - 实现对`AST`的遍历与语义分析、类型检查、符号绑定等。
- `midend`
  - `MidEnd.java`：中端统一接口，负责把`AST`转换为中间表示（`IR`），并调用`IR`层优化。
  - `llvm`包：`IrBuilder`、`IrModule`、`IrNode`、等，表示`LLVM`风格的中间表示与构建器。
- `optimize`
  - 多个优化和分析通道（`ActiveAnalysis`、`Lvn`、`MemToReg`、`InsertPhi`、`RemoveDeadCode/Block`、`RegisterAllocator`等），可组合用于中端 IR 优化与寄存器分配。

后端

- `backend`
  - `BackEnd.java`、`PeepHole.java`：后端调度与局部`peephole`优化。
  - `mips`：目标为 MIPS 的代码生成器（`MipsBuilder`、`MipsModule`、`Register`）以及细分的指令/汇编构造（`assembly`下的多个类），负责把`IR`翻译为`MIPS`汇编。
- `error`（错误处理）与`utils`（工具）
  - `ErrorRecorder/Error/ErrorType`：集中管理编译过程中的错误与诊断信息。
  - `utils`提供`IO`、调试、配置、复杂度处理等功能。

自己编译器设计

大致分为前、中、后三部分，按照词法分析、语法分析、语义分析、中间代码生成、目标代码生成、代码优化的顺序来写。

词法分析器负责生成`Token`流，语法分析器负责生成语法树。语义分析器解析生成的语法树，此部分参考上述编译器设计`Visitor`类分别对各语法成分进行分析。中间代码生成可能需要再设计另外的`Visitor`类结合`SymbolTable`来进一步分析。

目标代码生成器根据中间代码来生成，之后考虑代码优化部分。

一、文件组织(暂定)

```
├─frontend
|  └─Parser
|      ├─Decl
|      ├─Exp
|      ├─FuncDef
|      ├─MainFuncDef
|      ├─Stmt
|      ├─Token
|      └─Tree
└─midend
    ├─Symbol
    └─Visit
        ├─Decl
        ├─Exp
        ├─Func
        ├─MainFuncDef
        └─Stmt
```

二、词法分析设计

1. `token`类设计包含
    1. `token`枚举类型如`IDENFR`、`INTCON`、`CONSTTK`等；`TokenType type`
    2. 当前`token`对应字符串表示；`String lexeme`
    3. 当前`token`所在行号；
    4. ~~是否已经被输出；boolean isPrinted~~
2. 设计`Lexer`类包含
    - 源码文件流 `FileInputStream file`
    - 生成的tokens `ArrayList<Token> tokens`
    - 产生的错误 `ArrayList<Error> errors`
    - 当前读到的字符`currentChar`，当前读到的行数`currentLine`
3. 构造`Lexer`类时初始化源码文件流，并读取第一个字符。
4. 设计分析函数，可由`Complier`调用。

```
    public void analyse() {
        Token currentToken;
        while ((currentToken = getToken()) != null) {
            tokens.add(currentToken);
        }
    }
```

5. 分析函数通过调用`getToken()`函数每处理一个`token`就将其加入`tokens`数组中。
6. `getToken()`函数声明一个`StringBuilder lexeme`，根据自动机判断当前字符应该什么归于什么类型的处理函数。
    1. 首字符为英文字符或下划线 `_` ，进入处理标识符或关键字函数；
    2. 首字符为数字，进入整数常量处理函数；
    3. 首字符为引号 `"` ，进入字符串常量处理函数；
    4. 首字符为 `/` ，进入注释或除号处理函数；
    5. 首字符为 `+` `-` `*` `%` `;` `,` `(` `)` `[` `]` `{` `}` ，进入单字符运算符或分隔符处理函数；
    6. 首字符为 `=` `>` `<` `!` ，进入双字符运算符处理函数；
    7. 首字符为 `&` ，进入与符号处理函数；
    8. 首字符为 `|` ，进入或符号处理函数；
    9. `EOF`返回`null`；
7. 处理函数返回当前`token`。
8. 处理函数中出现的错误加入`errors`数组中，方便后续处理。

三、语法分析

1. 设计语法树：
    1. 一个语法成分对应一个结点，他们有公共的父类`Node`；
        - `Node`类包含枚举类`GrammarType type`表示当前语法成分的类型，如`Exp、IntConst`等；
        - `Node`类包含`Token token`，表示当前**终结符**对应的`Token`；
        - `Node`类包含`ArrayList<Token> tokens`，表示`Lexer`分析得到的`token`流；
        - `Node`类包含`int tokenIndex`，表示当前读入`token`流索引，进入/结束某一成分分析时`tokens[tokenIndex]`都应处于未处理状态；
        - `Node`类包含`Node parent`和`ArrayList<Node> children`分别表示父节点和子节点；
        - `Node`类包含`final String filename`和`final String ErrorFilename`分别表示正确输出文件和错误输出文件；
        - `Node`类含方法`Token peekToken(int offset)`，用于获取相对于索引`tokenIndex`偏移`offset`的值，可以预读或回读；
        - `Node`类包含方法`void printToError(Error error)`，输出对应错误时将其添加到`GlobalError`中；
    2. `Node`子类构造函数需`GrammarType type、int tokenIndex、ArrayList<Token> tokens`，终结符需设置`Token token`；
    3. 子类含分析方法，按照文法进行依次分析，调用对应成分分析方法前需：

1. 创建对象；
2. 添加为当前成分的子节点；
3. 调用`parser()`；

4. 分析方法末尾，先输出当前成分类型（终结符先输出当前`token`），再更新父节点的`tokenIndex`（终结符保留当前tokeIndex指向对应`token`，直接更新父节点`index+1`）；

2. 设计`Parser`类：
   - 包含`ArrayList<Token> tokens`表示传入的`token`流；
   - 包含`ComUnit root`表示初始根节点；
   - 函数`parser()`使用递归下降解析器，每个语法成分对应一个方法。
     - 进入新的语法成分时首先判断当前`token`属于接下来哪个非终结符；
     - 确定下一步要进入的语法成分对应的`parser()`函数后，首先创建此语法成分对象；
     - 然后将此对象加入到当前语法成分的子节点中，再调用子节点`parser()`；
     - 当前语法规则分析完毕后，更新父节点`tokenIndex`（终结符更新父节点为`index+1`，非终结符更新为`index`）；

```
    public void parser() {
        AddExp addExp = new AddExp(GrammarType.AddExp,
this.getIndex(), this.getTokens());
        this.addChild(addExp);
        addExp.parser();

        this.printTypeToFile();
        Node parent = this.getParent();
        parent.setIndex(this.getIndex());
    }
```

3. 对于类似`AddExp → MulExp | AddExp ('+' | '−') MulExp`的分析，由于存在左递归，将其转化成`MulExp { ('+' | '−') MulExp }`，但是位于运算符左边的成分实际上是`AddExp`而不是`MulExp`，所以如果存在运算符，应在左操作数调用`MulExp`的分析方法后输出成分`<AddExp>`，（仅用于输出，语法树的构造无需额外子节点`AddExp`）。

四、语义分析

1. 符号表设计：
   1. `SymbolType`类设计：

```
public enum SymbolType {
    CONST_INT("ConstInt"),
    CONST_INT_ARRAY("ConstIntArray"),
    STATIC_INT("StaticInt"),
    INT("Int"),
    INT_ARRAY("IntArray"),
    STATIC_INT_ARRAY("StaticIntArray"),
    VOID_FUNC("VoidFunc"),
    INT_FUNC("IntFunc"),
    ARRAY("Array"),
    NOT_ARRAY("NotArray"),
```

```
        NOT_EXIST("NotExist");
        private final String typeName;

        SymbolType(String typeName) {
            this.typeName = typeName;
        }

        public String getTypeName() {
            return typeName;
        }
    }
```

2. `Symbol`类设计：
    1. `String symbolName`表示符号名称；
    2. `SymbolType symbolType`表示符号类型：
    3. `int lineNumber`表示当前符号所在行；
    4. `ArrayList<Symbol> params`若当前符号为函数名，额外标记参数符号列表；
    5. `ArrayList<Integer>initValues`表示当前符号初始赋值；
    6. `int size`若当前符号为数组记录数组大小；
    7. `IrValue IrValue`，中间代码生成时使用，表示符号对应生成的`IrValue`；
3. `SymbolTable`类设计：
    1. `int depth`表示当前作用域序号（不是深度，只表示创建时顺序）
    2. `ArrayList<Symbol> symbolList`表示当前作用域中符号；
    3. `Hashtable<String, Symbol> symbolTable`表示当前作用域中符号方便查找；
    4. `SymbolTable fatherTable`表示父作用域对应符号表；
    5. `ArrayList<SymbolTable> sonTables`表示子作用域所含符号表；
    6. `int index`遍历所有符号表时，表示遍历到子作用域的子符号表索引；
4. 设计一个`GlobalSymbolTable`类作为全局作用域的符号表；
    1. `SymbolTable globalSymbolTable`表示全局作用域的符号表；
    2. 设计字段`SymbolTable localSymbolTable`表示当前处理的符号表；
    3. `int scopeDepth=1`表示作用域序号（不代表深度）；
5. 额外设计`OutSymbolTable`类，表示全局作用域之外的符号表，方便添加库函数；

2. 主`Visitor`类设计：
    1. `ComUnit comUnit`表示要分析的程序单元；
    2. `Visit()`方法遍历语法树；
        1. 首先通过`OutSymbolTable.addSymbol`添加库函数定义；

```
OutSymbolTable.addSymbol(new Symbol("getint",
SymbolType.VOID_FUNC,0,
        new IrFunction(ValueType.FUNCTION, IrType.INT32,
"@getint")));
```

        2. 依次遍历所有声明，函数定义，最后处理主函数；
3. `VisitorDecl`类设计：

1. `Decl → ConstDecl | VarDecl` 根据语法规则进行遍历；
2. 声明调用 `GlobalSymbolTable.addVarDef()` 或 `GlobalSymbolTable.addConstDef()` 添加进当前符号表；
    1. `GlobalSymbolTable.add...Def()` 方法
    2. 首先获取符号名，符号类型，当前行数创建 `Symbol` 对象；
    3. 然后依据符号类型判断是否设置数组长度；
    4. 为符号添加初始赋值（无初始值则保持空列表）；
    5. 添加符号进当前符号表；

```
public static void addVarDef(VarDef varDef, boolean isStatic) {
    String Ident = varDef.GetIdent();
    SymbolType symbolType = varDef.GetSymbolType();
    ...//静态变量类型转换
    int line = varDef.GetLineNumber();
    Symbol symbol = new Symbol(Ident, symbolType, line);
    if (symbolIsArray(symbolType)) {
        symbol.setSize(varDef.GetArraySize());
    }
    symbol.setInitValues(varDef.GetInitValues());//为符号添加初始赋值（无初始值则保持空列表）；
    localSymbolTable.AddSymbol(symbol);
}
```

3. `VisitorFuncDef` 类设计：
    1. 获取返回值类型标记接下来要处理的块为最外层 `Block`；
    2. 作用域加一，访问函数参数，回到上一级作用域，再添加符号到符号表
    3. 进入下一级作用域，访问 `Block`，访问结束回到上一级作用域；
    4. `main` 函数处理类似，但是不添加符号到符号表；
4. `VisitorStmt` 类设计：
    1. `stmt.isBlock()||stmt.isIfStmt()||stmt.HasElseStmt()||stmt.isForStmt()` 分别调用对应的 `Visitor` 类；
    2. `stmt.isLVal()`，调用 `VisitLVal()` 和 `VisitExp()`；设计 `VisitorLVal` 类，可能出现的错误：
        1. `LVal` 为常量时，不能对其修改；`ErrorType.h`
        2. 变量未定义；`ErrorType.c`
    3. `stmt.isExp()` 调用 `VisitExp()`；设计 `VisitExp()`，可能出现的错误：
        1. 函数未定义；`ErrorType.c`；
        2. 函数调用时无实参，但函数定义有形参;`ErrorType.d`
    4. `stmt.isReturn()` 调用 `VisitReturn()`；可能出现的错误：有返回值但不需要 `ErrorType.f`；
    5. `stmt.isBreakContinue()` 调用 `VisitJump()`；
        1. 在语法分析阶段设置变量 `boolean isForBody`

2. 可能出现的错误：非循环块使用 `break` 和 `continue`； `ErrorType.m`；

6. `stmt.isPrintf()` 调用 `VisitPrint()`；可能出现的错误： `%d` 数量不匹配； `ErrorType.l`

五、中间代码生成

采用 `LLVM` 作为中间代码

IrType设计

`IrType` 用来描述 `IrValue` 的类型；

与 `enum` 相似，通过 `typeName` 字段来标识不同类型；

```
public static final IrType MODULE = new IrType("module");
public static final IrType FUNCTION = new IrType("function");
public static final IrType POINTER = new IrType("pointer");
public static final IrType BASICBLOCK = new IrType("basicblock");
public static final IrType VOID = new IrType("void");
public static final IrType INT1 = new IrType("i1");
public static final IrType INT8 = new IrType("i8");
public static final IrType INT32 = new IrType("i32");
public static final IrType ARRAY = new IrType("array");//array元素默认int32
public static final IrType STRING = new IrType("string");//string元素默认int8
private final String typeName;
```

额外添加 `int arraySize` 来表示数组类型的数组长度；

对于指针类型 `IrType("pointer")`，设计子类 `IrPointer`，包含 `public IrType targetType;` 表示指针指向的类型。如 `IrGlobalValue` 的类型为 `new IrPointer(initial.irType)`；

注意：对于 `array` 和 `string` 类型需要创建新的IrType类而不是直接使用IrType.ARRAY来赋值，因为不同的数组需要赋arraySize的值不同需要新的对象

为方便处理循环体，额外设计 `IrLoop` 类，包含四个基本块

```
private IrBasicBlock condBlock;
private IrBasicBlock bodyBlock;
private IrBasicBlock stepBlock;
private IrBasicBlock afterBlock;
```

IrValue设计

我对 `IrValue` 的理解：对于源程序有语法树，各语法成分有公共父类 `Node`，而 `IrValue` 有点像是对于 `LLVM` 程序的 `Node` 类；

LLVM由 `IrValue` 组成，生成 `LLVM IR` 的过程就是构造各种 `IrValue` 的过程；

```
public final ValueType valueType;// 枚举类（仅用来标识）
public final IrType irType; // 对于变量/常量表示其类型，对于函数/指令表示返回值类型
public final String irName; // LLVM中可能会使用的命名
public final ArrayList<IrUse> useList; // 记录当前IrValue所包含的使用关系（只有被使用者才有内容）
```

`IrUse`类表示一种使用的关系，即记录了谁使用了谁；

`IrUser`类继承`IrValue`类，表示使用者，包含`ArrayList<IrValue> useValues`表示其使用的值，这也是为什么使用者的`useList`没有添加内容。

Instruction类

所有指令的父类，继承`IrValue`类

`InstructionType instrType`表示指令的类型，`IrBasicBlock inBasicBlock`表示当前指令所在的基本块；

```
// 添加指令到当前基本块
public static void addInstr(Instruction instr) {
    currentBasicBlock.addInstruction(instr);
    instr.setParentBasicBlock(currentBasicBlock);
}
```

指令构造函数中并没有设计自动添加基本块，所以需在`IrBuilder`中显式地为指令添加基本块。

IrBasicBlock类

基本块类，同样继承`IrValue`类

`ArrayList<Instruction> instructions;`按顺序记录基本块所含指令；

`IrFunction function;`表示此基本块所属函数（此处函数有点类似于作用域的含义）；

IrModule类

与`comUnit`类似，表示`LLVM`的整体。

包含：

```
ArrayList<String> declares; // 定义
ArrayList<IrFunction>functions; // 函数
ArrayList<IrGlobalValue> globals; // 全局变量/静态变量
HashMap<String, IrConstString>stringIrConstStringHashMap; // 字符串常量
```

IrBuilder设计

`IrModule irModule`表示输出的`LLVM IR`单元；

`IrFunction currentFunction`表示当前正在处理的函数；

`IrBasicBlock currentBasicBlock`表示当前正在处理的基本块；

`Stack<IrLoop> loopStack`用来记录循环体的嵌套；

`int basicBlockNum`、`int globalVarNum`、`int stringConstNum`分别记录基本块、全局/静态变量、字符串常量的数目方便命名；

`HashMap<IrFunction, Integer> localVarNum`记录每个函数内的局部变量数目；

所有`Instruction`指令类型的对象都通过`IrBuilder`中`GetNew...()`方法来创建；

生成过程

与语义分析过程相似，依次遍历所有声明，函数定义，最后处理主函数。与语义分析添加符号到符号表对应的是，在中间代码生成阶段为`Symbol`类中`IrValue IrValue`字段赋予对应的对象。最后重写各`IrValue`的`toString()`方法并依次输出；给出`IrModule`的`toString()`方法：

```
public String toString() {
    StringBuilder sb = new StringBuilder();
    //输出声明
    for (String declare : declares) {
        sb.append(declare).append("\n");
    }
    //输出字符串常量
    List<Map.Entry<String, IrConstString>> stringEntries
        = new ArrayList<>(this.stringIrConstStringHashMap.entrySet());
    for (Map.Entry<String, IrConstString> entry : stringEntries) {
        sb.append(entry.getValue()).append("\n");
    }
    //输出全局变量
    for (IrGlobalValue global : globals) {
        sb.append(global).append("\n");
    }
    //输出函数
    for (IrFunction function : functions) {
        sb.append(function).append("\n");
    }
    return sb.toString();
}
```