BUAA-2025-Compiler 设计文档

一、参考编译器介绍

我主要参考了Hyggge's Blog的编译器设计博客以及zhangyitonggg的编译器代码，这让我对所要实现的编译器功能、框架、模块有了清楚的认识，帮我更好的设计我所要实现的编译器。下面我以zhangyitonggg的编译器进行简单介绍：

1.1 总体设计

参考编译器使用Java语言编写，中间代码形式为LLVM，目标代码为Mips.。编译器整体设计分为前端(词法、语法、语义分析)&&中端(LLVM生成、Mem2Reg优化)&&后端(MIPS生成，寄存器分配)

1.2 接口设计

编译器的词法分析、语法分析、语义分析、IR生成、MIPS生成均采用单例模式设计，各个模块间接口如下

阶段	输入	输出	核心方法
词法分析	testfile.txt	tokenStream	lex()
语法分析	tokenStream	CompUnit(AST根)	parse()
语义分析	CompUnit(AST根)	SymbolTable	check()
IR生成	SymbolTable	Module	visit()
目标代码	Module	`String` (MIPS)	map()

1.3 文件组织

```
src/
├── Compiler.java           # 主入口，串联编译流程
├── frontend/               # 前端
│   ├── lexer/              # 词法分析器
│   │   ├── Lexer.java      # 词法分析主类（单例模式）
│   │   ├── TokenStream.java # Token流
│   │
│   ├── parser/             # 语法分析器
│   │   ├── Parser.java     # 语法分析主类（递归下降）
│   │   └── AST/            # AST节点定义
│   │
│   └── checker/            # 语义分析器
│       ├── Checker.java    # 语义检查主类
│       ├── SymbolTable.java # 符号表
│
├── llvm/                   # 中间代码（LLVM IR）
│   ├── Visitor.java        # AST遍历生成IR
│   ├── IrFactory.java      # IR指令工厂
│   ├── Module.java         # IR模块（顶层容器）
│   ├── types/              # IR类型系统
│   └── value/              # IR值
│
```

```
├── optimize/              # 优化模块
│   ├── Mem2Reg.java       # Mem2Reg优化（将栈变量提升到SSA）
│   ├── RegAlloca.java     # 寄存器分配
│   └── RemovePhi.java     # 消除Phi节点
│
├── backend/               # 后端（MIPS生成）
│   ├── Mapper.java        # IR到MIPS映射主类
│   └── Instruction/       # MIPS指令类
│
└── Utils/                 # 工具类
    ├── Printer.java       # 输出工具
    ├── Error.java         # 错误类
```

二、编译器总体设计

2.1 总体结构

编程语言采用`Java`,中间代码`LLVM`,目标代码生成`mips`。系统架构也采用前中后三端架构

- 前端：词法分析、语法分析、语义分析。主要进行错误处理与生成语法树
- 中端：生成LLVM中间代码，并通过Mem2Reg指令将中间代码转换为`SSA`形式。进行死代码删除、常量计算
- 后端：寄存器分配、目标代码生成

2.2接口设计

词法分析、语法分析、语义分析、IR生成、MIPS生成五大核心模块采用单例模式类。在每一个单例模式类中完成各个语法成分的转换。例如在语义分析单例类`checker.Java`中通过`checkCompUnit()`、`checkDecl()`等方法检查各个语法成分是否有错误。

对于代码优化，我设计了一系列工具类来进行对应种类的优化。例如`GVN`工具类进行公共子表达式删除

2.3 文件组织

```
compiler_sysy/src/                          编译器源代码
|   ├── 📄 Compiler.java                    ⭐ 编译器主程序入口
|   ├── 📄 config.json                      编译器配置
|   |
|   ├── 📁 frontend/                        前端（词法/语法/语义）
|   |   ├── Lexer/
|   |   |   ├── Lexer.java                  词法分析器
|   |   |
|   |   ├── Parser/
|   |   |   ├── Parser.java                 ⭐ 语法解析器
|   |   |   └── AST/                        抽象语法树定义
|   |   |
|   |   └── Checker/
|   |       ├── Checker.java                ⭐ 语义检查
|   |       ├── SymbolTable.java            符号表管理
|   |
|   ├── 📁 llvm/                            中端（LLVM IR）
|   |   ├── IRBuilder.java
|   |   ├── Visitor.Java                    ⭐ IR 构建器
|   |   ├── LlvmModule.java                 LLVM 模块
```

```
|   |   ├── Value.java                    值基类
|   |   ├── type/                         类型系统
|   |   ├── instruction/                  IR 指令
|   |   └── UserClass/                    自定义类型
|   |
|   ├── 📁 optimize/                      优化（中端）
|   |   ├── preOptimize.java              常量折叠、代数恒等式
|   |   ├── Mem2Reg.java
|   |   ├── DeadCodeElimination.java
|   |   ├── RegAlloca.java
|   |   └── RemovePhi.java                Phi 节点消除
|   |
|   ├── 📁 backend/                       后端（MIPS 代码生成）
|   |   ├── MipsGenerator.java            ⭐ MIPS 代码生成器
|   |   ├── MipsBuilder.java              ⭐ MIPS 指令构建
|   |   ├── MipsModule.java               MIPS 模块
|   |   ├── Reg.java                      寄存器管理
|   |   |
|   |   ├── Data/                         数据定义
|   |   └── Instruction/                  MIPS 指令
|   |
|   └── 📁 Utils/                         工具模块
```

三、词法分析

设计

> 此法分析主要采取**一符一类**实现(特殊字符和保留字单独成类)

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	else	ELSETK	*	MULT	;	SEMICN
IntConst	INTCON	!	NOT	/	DIV	,	COMMA
StringConst	STRCON	&&	AND	%	MOD	(	LPARENT
const	CONSTTK	||	OR	<	LSS	)	RPARENT
int	INTTK	for	FORTK	<=	LEQ	[	LBRACK
static	STATICTK	return	RETURNTK	>	GRE	]	RBRACK
break	BREAKTK	void	VOIDTK	>=	GEQ	{	LBRACE
continue	CONTINUETK	+	PLUS	==	EQL	}	RBRACE
if	IFTK	-	MINU	!=	NEQ	=	ASSIGN
main	MAINTK	printf	PRINTFTK				

词法分析主要由`frotend/Lexer/`下面的文件完成，利用`FileReader`类进行字符读取，利用`FileReader`

的`read()`与`unread()`方法实现字符读取与回退

`Lexer.java` 负责词法分析的主要逻辑，对外提供 `public TokenStream lex();` 方法，返回tokenList给语法分析模块; (`TokenStream` 类中只存储一个`ArrayList`， 重写`toString()` 方法)

识别出的`Token` 封装为如下`Token`类，并重写`toString()` 方法，用于`lexer.txt` 输出

```
public class Token {
    private TokenType type;
    private String value; // 对于标识符、整数、字符串常量，存储其值
    private int lineNumber;   // 行号
}
```

修改

> **文件读取结束的判定**
> FileReade `read()`函数r在读取到文件结束符EOF时会返回 -1, 但是如果随意将代表EOF的 -1 进行`unread()`, 会导致`-1` 被错误地当作普通字符(不是EOF)存入回退缓冲区，之后再利用`read()`进行读取时,原先存入的`-1`会返回`65535` 而不是`-1`,可能导致程序无法正确判断读入是否结束
>
> 且读取到文件结尾时，只要不对EOF进行`unread()` 操作，那么之后`read()`得到的均为`EOF(-1)`

四、语法分析

语法分析模块主要由`src/frontend/Parser/Parser.java`完成，其输入是词法分析输出的单词流`tokenstream`，按照文法进行递归下降分析，构造出AST语法树。

AST语法树类

我们为每一种语法成分均建立对应的节点类，并且所有语法成分类均继承父类`Node`。对于类型相近的语法节点类，我们为其设立了接口，便于节点管理。将语法树节点类也可以分为以下几类：

- **Exp**：主要有`AddExp`、`MulExp`、`ConstExp`、`EqExp`、`UnaryExp`、`Exp`等表达式相关类
- **Stmt**: 我为`BlockStmt`、`BreakStmt`、`IfStmt`等`stmt`相关产生式设计了语法树节点类，均继承自`Stmt`接口，便于`stmt`语法的解析。
- **Func**：包括`FuncDef`、`FuncFParam`、`FuncFParams`、`FuncRParams`等类
- **Decl**: 包括`VarDecl`与`ConstDecl`等类

> 每一个语法树节点类中存储该项语法成分的组成元素

递归下降分析

在递归下降之前，我们需要消除原文法中的左递归，例如

```
/**
 * LOrExp → LAndExp | LOrExp '||' LAndExp
 * 去除左递归 LOrExp → LAndExp {'||' LAndExp}
 */
```

之后我们为每一个语法成分编写`parse()`方法，按照递归下降原则进行调用解析

```
public CompUnit parseCompUnit() {\
    // ...
    }
```

```
    // Decl → ConstDecl | VarDecl
    public Decl parseDecl() {
    // ...
    }

    // ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';' // i
    public ConstDecl parseConstDecl() {
    // ...
    }
```

回溯处理

在解析Stmt时候我们会遇到一个问题`Stmt → LVal '=' Exp ';'`与`Stmt→[Exp] ';'`并不能简单通过`first`集合判断应该采取那一条产生式。此处需要采用回溯分析

- 监视点记录当前token流位置，调用`parseExp()`
- 解析`Exp`之后，根据其后是否出现`=`号选择对应产生式，并重置tokenstream的观测点为之前记录的监视点

错误处理

主要检语法错误，包括缺少分号、缺少右小括号、缺少右中括号等

修改

实际上我在Node中并没有设置通用属性，故将`Node`节点改为接口类，以表征各个语法树节点类在构造语法树过程中的行为一致性。

语义分析

在语义分析阶段，我们需要构造符号表，在其中记录变量等的作用域、类别、名称等信息，并借助符号表进行语义相关的错误处理

文件组织

- 符号类：`Symbol.java`,其中属性有枚举变量(`SymbolType`)与(`name`)。其子类有`ConstSymbol`,`FuncSymbol`,`VarSymbol`等，在子类属性中记录不同类型`symbol`的独有属性
- 符号表：`SymbolTable`
    - ```
        private int id;   // 作用域序号
        private ArrayList<Symbol> symbols;      // 存储当前作用域下的symbol
        private SymbolTable parent;     // 存放父作用域的符号表，最外层作用域无父作用域
        private ArrayList<SymbolTable> children;     // 存放当前作用域的直接子作用域的符号表
      ```

符号表构造与错误处理

工作流程与语法分析类似，只不过为了各个模块间耦合性降低，我们将其分在两处实现。

具体来说是为每一个语法成分编写`check()`方法，`check()`间通过递归下降调用分析各个符号。
      ```

修改

为每一个`check()`方法添加对应的文法规则注释

中间代码生成-LLVM

我选择以LLVM作为中间代码。主要原因是想要借此机会深入了解学习一下这种编译器中常用的中间代码形式

文件组织

主要有以下几类：

- **Value**类：LLVM中有一句话是一切皆`Value`，这主要是因为LLVM语法中所有的类均继承自Value。
- **User**类：User类同样继承自Value，同时其属性中含有Value，代表该User使用的Value有哪些。这其实也隐含了`user-use`关系。`Function、BasicBlock、Instruction`也均继承于此类
- **LLVMType**类，具体来说有`ArrayType、FuncType、LabelType、PointerType、VoidType`等
- **Module**类：是我们LLVM代码生成最终构建的目标，管理着所有`Value`
- **Visitor**类：单例模式类，在其中撰写针对于各个语法成分的翻译方法
- **IRBuilder**：在其中实现各个`LLVM-Instruction`的构造方法，便于`visitor`调用

工作流程

- **循环栈的设置**：在翻译到`break`等时，我们需要明确LLVM生成的跳转指令的目标基本块是谁。但是由于嵌套循环的存在，我们不可直接从某一个静态变量或者属性中得到目标块。需要设定循环栈，在遇到新循环时候，将其`initBb, condBb, bodyBb, followBb, endBb`等压栈，处理完成后弹栈。
- **关于`Lval`的处理**：对于Lval来说，在赋值表达式中出现时，我们需要取出其地址(指针）。在其他地方需要取出其值。按照这样的分类方法，可减少i`f-elseif-else`数量

编码完成之后的修改

编码完成后，我意识到常量计算完全可以在该阶段计算完成，我在这里添加了常量传播相关代码。

目标代码生成-Mips

我选择了Mips作为目标代码，主要是为了参与竞速排序

文件组织

- **Data**类：其类别与`mips`中的`.data`段类似
- **Instruction**：类别与**mips**的各个指令相同
- **Reg**：管理通用寄存器与惩处寄存器
- **MipsGenerator与MipsBuilder**：主要用于将LLVM中的各类指令翻译为mips指令
- **MipsModule**：管理最终生成的mips程序的`.data`段与`.text`段

工作流程

(此处工作主要在优化之前)

由于**LLVM**语言相对来说也是一种底层语言，其与**mips**之间的跨度并不是很大。所以我的主要处理方法是为LLVM的每一条指令编写了一个`translate`方法

为了快速完成mips生成，我并没有考虑优化。而是将所有变量均存在了栈空间上。

修改

主要是在代码优化时添加了寄存器分配

代码优化

我主要进行了以下代码优化：

死代码删除

在死代码删除方面，我主要进行了三个维度的代码优化。

- 不可达的基本块：从每一个函数的**entryBb**开始，将基本块视作节点，`jump`或者`branch`指令视作两个基本块之间的有向边进行dfs遍历，删除未到达的基本块
- 未调用的函数：类似于基本块删除，将各个函数视作节点，函数调用视作节点之间的边，进行dfs遍历。删除不可到达的函数
- def-use分析：根据定义-使用链删除未使用的load指令等

Mem2Reg

LLVM通过alloc-load-store将所用变量存储在内存空间中，这会增加内存访问的时间。Mem2Reg优化主要将变量存储在寄存器中，并通过phi指令将LLVM转换为SSA形式。

在removePhi的过程中，我们将各个phi指令转换成move指令。不可避免的，在这个过程中会引入很多move指令，这依赖于良好的寄存器分配策略去提升其性能。很遗憾的是我的寄存器分配比较糟糕

寄存器分配

我采用的是基于引用计数的寄存器分配策略

- 遍历各个指令统计各个值的使用次数与生存周期，为使用次数多的值赋予较高的权重，减少不必要的内存访问。
- $v1,$t0-$t9, $s0-$s7，$fp等寄存器参与寄存器分配

其他

乘除优化、常量传播、简单的窥孔优化等

总结

整个编译程序最后的代码行数超过了万行，是我所写过的最大的一个项目，对于我的架构设计能力提升巨大。