编译原理课设设计文档

author: 张奕彤 22373337

零、前言

2024年09月06日

2024年暑假曾经参与过一段时间的编译比赛，但由于队内成员大多有科研实习的任务，所以在写完语义分析后也就放弃比赛了。虽然放弃这个比赛让我获得了一些实习的成果，但终归还是有些许遗憾的。希望在编译课上，我能够凭借自己写出一个让自己满意的编译器。

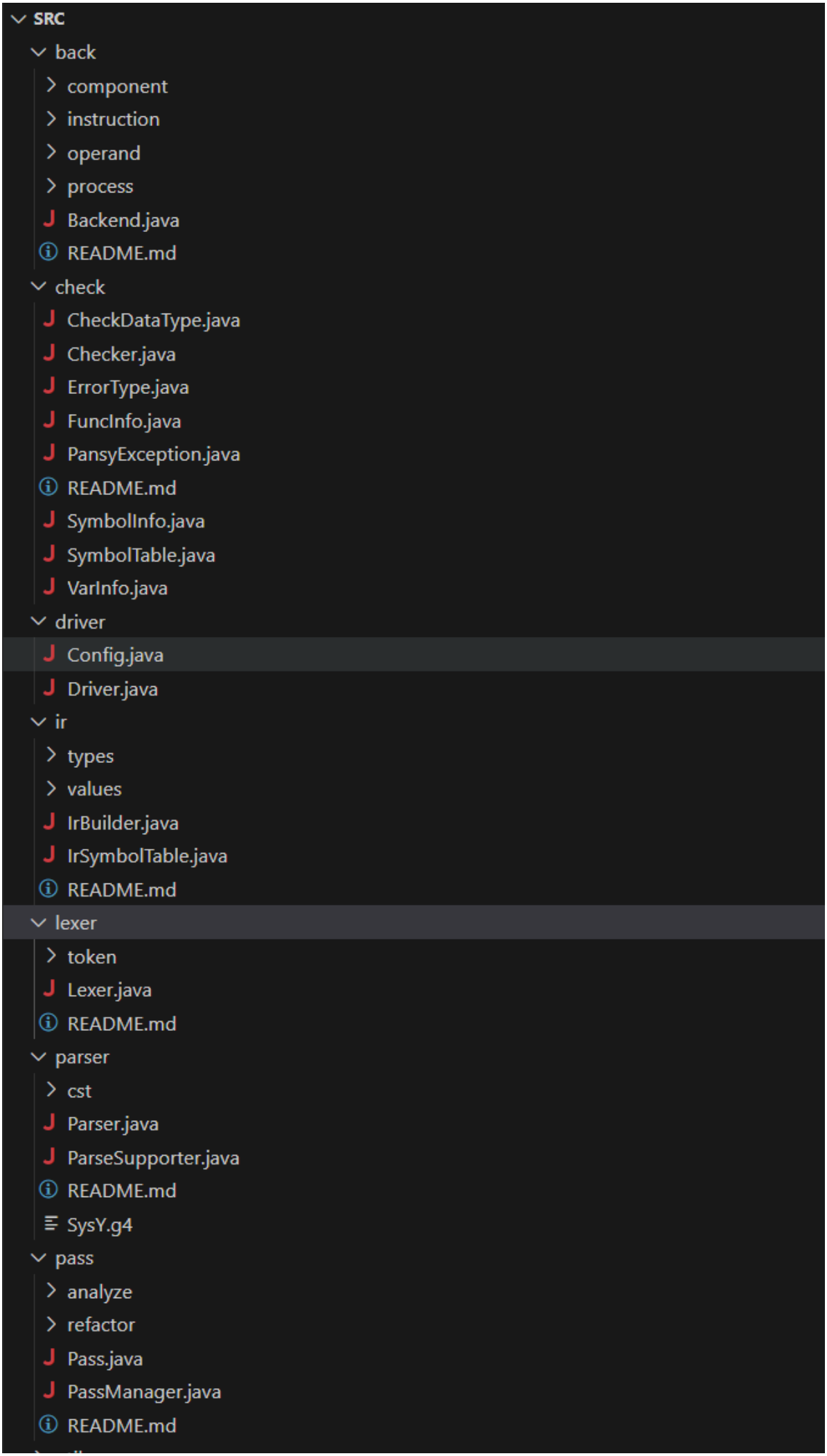
2024年12月06日

之前想到这学期会在科研和陶瓷中花费不少时间，现在看来甚至我低估了做这些事所需要花费的时间和精力，写出的编译器也没有让我满意，最终草草收尾。

一、参考编译器介绍

主要参考了hyggge的代码、qsgg写的代码、龙书、北航自己写的《编译原理》以及系统能力大赛中开源的一些代码，如喵喵队引体向上。

这些参考编译器大多是采取了 LLVM 作为中间代码，并整体分成了前端、中端、后端三个部分。下面是qsgg代码的文件组织：


SRC
back
component
instruction
operand
process
Backend.java
README.md
check
CheckDataType.java
Checker.java
ErrorType.java
FuncInfo.java
PansyException.java
README.md
SymbolInfo.java
SymbolTable.java
VarInfo.java
driver
Config.java
Driver.java
ir
types
values
IrBuilder.java
IrSymbolTable.java
README.md
lexer
token
Lexer.java
README.md
parser
cst
Parser.java
ParseSupporter.java
README.md
SysY.g4
pass
analyze
refactor
Pass.java
PassManager.java
README.md

二、编译器总体设计

总体结构与接口

计划选择 Java 作为开发语言，并采取 LLVM 作为中间代码，最终计划生成 mips 汇编代码。该编译器将分为前端、中端、后端三部分。

- 前端包括词法分析、语法分析、语义分析。
  - 词法分析将以文件为输入，以 TokenStream （即 Token 的列表）为输出。
  - 语法分析将以 TokenStream 为输入，递归构建抽象语法树 AST，返回 AST 的根节点。
  - 语义分析将以 AST 的根节点为输入，递归地生成整个抽象语法树的 LLVM IR,并返回 LLVM IR 的顶层模块 module。
- 中端包括中间代码优化。
  - 中间代码优化以 module 为输入，进行若干种代码优化的操作，然后还是返回 module。
- 后端包括目标代码生成。
  - 目标代码生成以进行完中间代码优化的 module 为输入，对 LLVM IR 递归地进行目标代码生成，最终返回 mips 指令序列。

文件组织

以 Compiler.java 中的 main 方法为程序的入口，main方法也将负责词法分析、语法分析等五个部分之间的衔接。

项目还包含如下几个软件包：

- frontend：即前端部分，负责词法分析、语法分析、错误处理。
- llvm：负责语义分析，生成中间代码。
- optimize：负责进行主要的优化，主要有mem2reg、removePhi和基于线性扫描的一个比较简单的寄存器分配。
- backend：即后端部分，负责目标代码生成。
- Utils： 内含多种工具，比如存储错误信息的 ErrorLog。

三、词法分析设计

文件组织

词法分析主要由四个类进行，分别是 Lexer, Token, TokenStream, TokenType。

- Lexer：负责对输入文件进行词法分析。
- Token：存储 Token 的类别、值、行数这些基本信息。
- TokenStream：由若干 Token 组成，是词法分析最终要得到的 对象。
- TokenType：枚举类，内含编译实验课程要求的所有类别。

工作流程

Lexer 类的 lex 方法会不断读取输入文件中的字符，如果读取到了文件结尾，则将此时的 TokenStream 对象返回；如果读取到了换行符，则将行数加一，并继续向后读入一个字符；否则，则调用 findToken 方法向后读取一个 Token 并加入 TokenStream 对象。

findToken 的逻辑较为清晰，大体如下：

- 读到双引号时，则读取从此双引号到下一个双引号为止的所有字符，此字符串为 STRCON 类。
- 读到单引号时，则根据下一个字符是否是反斜杠来决定向后读一个字符还是两个字符，最终读取到的值即为 CHRCON 类。
- 读到数字时，则继续向后读取，直到读到非数字字符为止。读取到的值即为 INTCON 类。
- 读到下划线、字母时，则继续向后读取，直到读到不是下划线、字母、数字的字符为止。然后将读取到的值与保留关键字表进行比对，如果属于该表，则说明识别到了一个保留关键字，否则则是一个标识符。
- 读到 `! & | / < > =` 时，需要向后多读取一个字符，从而判断是何种 Token。
- 读到 `+ - * % ; , ( ) [ ] { }` 时，可以直接判断当前读取到这个字符是何种 Token。
- 读到 `//` 时，则向后继续读，直到读取到换行符为止，当前读取到的值（不含换行符）为单行注释。
- 读到 `/*` 时，则向后继续读，直到读取到 `*/` 时为止，当前读取到的值为多行注释。

错误处理

词法分析中的错误处理非常简单，只需要再遇到 `$` 和 `|` 后检查一下个数即可。

编码完成之后的修改

- 在处理多行注释时，遇到换行符时没有将行数加一，造成了行数不对的错误。
- 第一次编码完成之后，发现所有类都集中在 frontend 包中，感觉过于臃肿了，于是将记录错误的 Error 和 ErrorLog 类移到了 Utils 包中。

四、语法分析

语法分析的主要任务是将词法分析生成的单词流（TokenStream）转化为一棵抽象语法树（AST），为后续的语义分析和代码生成提供结构化的数据。

文件组织

主要由AST中的各个节点类和Parser组成。

- Parser：负责对词法分析得到的token流进行分析。
- AST：AST中包含了各个节点。

工作流程

这里就需要用到在你6面向对象课程中所学的递归下降法了。我们首先改写文法，消除其中的左递归，然后我们在各个节点类中按照改写后的文法定义各个节点的属性和相应的Get和Set方法。之后，我们在Parser类中递归向下分析每个节点类即可。具体细节如下：

1. **递归下降解析**
   每种语法规则对应一个解析函数。例如：
   - `parseAddExp()` 用于解析 `AddExp`，它递归调用 `parseMulExp()`。
2. **回溯支持**
   - 在可能的多分支选择中，通过预读下一个 Token 判断选择路径。
   - 如果无法通过预读解决选择路径的话，则需要进行回溯，此时仅需要记录下回溯返回的位置，并在结束时返回合适的位置即可。

错误处理

这里的错误处理类别也比较少，主要有缺少分号、缺少右小括号、缺少右中括号这么几个情况。

此处可以使用我们课上所说的First集和Follow集的概念进行解决。

编码完成之后的修改

当我写完语法分析，我突然意识虽然那我确实给每一个节点类设计了一个公共的父类，但是这个父类似乎没有发挥任何作用，于是做了如下修改：

- 在所有公共的父类中增加了一些通用属性（如节点所在行号）。
- 子类继承父类并扩展具体语法节点的功能。

这样就让我的代码在一些具体实现上变得优雅了许多（虽然还是很ugly）。

语义分析

语义分析主要需要做两件事：

- 对于**正确的源程序**，需要从源程序中识别出定义的常量、变量等，输出它们的作用域序号，单词的字符/字符串形式，类型名称。
- 对于**错误的源程序**，需要识别出错误，并输出**错误所在的行号**和**错误的类别码**。

可以看到课程组有意识地将错误处理分散在编译器的全流程中，事实上这确实是合理的，因为真正的编译器不可能是只在考察编译器错误处理能力时提供错误的代码。

文件组织

主要由三部分组成

- 符号类Symbol：存储一个常量 or 变量 or 函数声明 or 其他的一些必要的信息。它两三个子类：
  - VarSymbol
  - FuncSymbol

  在此处，我将文法中所谓的常量和变量全都放在了VarSymbol中。因为按照我们亲爱的荣文戈老师的说法，C语言标准中是没有常量和变量的提法的~~（欢迎选修C语言系统级编程）~~。

  除此之外，我还涉及了一个SymbolType类，给每一个Symbol分配一个type。
- 符号表类SymbolTable：存储当前作用域下的所有符号，并存储父作用域的符号表和所有子作用域的符号表。
- Checker类：进行语义分析的主体。

工作流程

我感觉这里和语法分析非常像，无非是从顶层模块开始往下递归的分析，只不过语法分析时在分析的过程中会把节点给创建出来，而这里我们需要分析的对象即是语法分析中得到的节点。

这里有一些难度的就是检查“在非循环块中使用continue和break”块，我在Checker类中定义了一个记录循环深度的属性，用于判断break和continue是否合法。

编码完成之后的修改

第一次写完后我意识到有很多方法只会在类内使用，如果还是设为public的话在迭代开发时会有不小的麻烦。因此我将语法分析和语义分析中的所有仅在类内使用的方法改为了private。

中间代码生成

这里将中间代码生成单独拆出来讲。

我是选择的LLVM作为中间代码，原因主要有三：

- 有很多往年学长的博客、代码可以参考，便于实现；
- LLVM本身是十分易于优化的。
- 即使最后完不成mips代码生成，也可以选择LLVM作为目标代码，可谓是进可攻、退可守。

文件组织

这里设计的文件比较多，主要有如下几类：

- IrType：为每一个Value分配一个类型，有ArrayType、DataType、FuncType、LabelType、PointerType、VoidType，它们共同继承自Type类。
- IrSymbolTable：这里我新建了一个符号表，而没有使用之前语义分析时的符号表，可以说是非常遗憾的一个地方。
- Value：内含许多Value类。
- Module：管理所有Value。
- Use：记录User-Used关系。
- IrFactory：提供一个构建Value的工厂。
- Visitor：递归地分析每个语法节点。

工作流程

之前我曾经与fzy讨论过中间代码生成时到底是要做什么，当时我给出的答案是：首先为每个节点创建不同数量不同类型的Value，其次则是将这些Value插入Module中特定的位置。

将所有逻辑放在Visitor中可以使得整个流程清晰可控，却带来了一个非常大的问题，那便是代码中有很多细节的处理，变得十分的冗长，也就是太不抽象了。为了对抗这种复杂性，我们引入了IrFactory，将创建类的具体细节和将对应的Value插入Module中对应的位置的逻辑放在了IrFactory中去做。

值得一提的是，诸如常量的计算是完全可以在这个阶段去做的，因此我在这里对常量进行了传播。

错误处理

对于错误的程序，会最晚在语义分析阶段就会终止，因此此处的错误处理没有任何新增的部分，本节以及下文也便不再赘述了。

编码完成之后的修改

在我debug的过程中，我发现一千多行的Visitor类很难准确定位bug，因此我完整地阅读了一遍我的代码，为每一个逻辑加入了对应的注释。

目标代码生成

我选择了Mips作为目标代码，下面是相关实现细节。

文件组织

- Instruction：其中的类对应Mips中具体的指令。
- Data：其中的类对应Mips中的各种data段的数据。
- Reg：内部管理了32个通用寄存器和Hi、Lo两个寄存器。
- MipsFactory：与IrFactory类似。
- Mapper：与Visitor类似。

工作流程

我在暑假时曾经了解到了为每个数据或者指令分配虚拟寄存器的方法。当时感觉非常便于优化，但是真到了自己去写Mips代码生成的时候，突然觉得没有那么多时间让我去思考了，因此就野蛮地直接分配起了物理寄存器，这也使得我的代码变得非常的丑陋：随处可见的if、随处可见的特例……

这里将主逻辑全部封装在了Mapper类中。为什么取这个名呢，因为我认为LLVM是一种十分偏向底层的语言，与Mips几乎是一一对应的关系，因此从LLVM生成Mips的工作与其说是翻译，不如说是映射。实际上也确实如此，在实现目标代码生成时，我几乎都是顺序地读入中断的Value，然后将其映射到某一个或多个Mips指令。

编码完成之后的修改

编码的过程是十分简答的，大约只用了一天也就写完了。但是编码完成之后debug却花了不少时间。这是因为Mips代码十分地不抽象，将所有细节都暴露出来了，因此我引入了注释类，来将LLVM代码以注释的形式加入到对应的一节Mips代码前。

代码优化部分

编码前的设计

由于本人能力有限、经历有限，在这里我只做了三个最基础的优化。

mem2reg

mem2reg 优化主要用于将内存访问转换为寄存器访问。LLVM 中的许多操作需要读取内存中的值，可以通过此优化将内存中的值加载到寄存器，从而减少内存访问的开销。在实现时，通过遍历每个基本块中的 load 和 store 操作，将其删除，并更新相关的指令以及引入Phi即可。

removePhi

这里姑且将removePhi算作优化的一部分吧。毕竟如果没有这一步，那mem2reg也就很难发挥作用。

这里我是完全参考的编译教程中的removePhi的方法，即将Phi指令转化为若干copy指令，并适当引入新的copy指令来消除寄存器共享和并行效果被破坏的情况。

基于引用计数的寄存器分配

- 在生成目标代码时，遍历变量的引用计数，并尽可能将频繁使用的变量分配到寄存器，而将较少使用的变量存储到内存中，从而减少不必要的内存访问操作。
- 实现时，首先需要构建一个引用计数的HashMap，记录每个变量在程序中的使用情况，并通过此信息进行寄存器分配的优先级排序。

比较遗憾的是这里我没有考虑循环中的变量更有资格获得寄存器的情况，不过效果已经不错了就是了。

编码完成后的修改

这里的修改主要是在mem2reg中，第一次编码完成后，我发现我的代码生成二和竞速排序有几个点过不了，后来通过利用gpt生成测试样例发现是地址对齐的问题。

总结

终于是走到了最后，本来想长篇大论的，但是现在却有没有任何话想说，只想赶快忙完手头的活，然后倒头睡去。