申优文档

79066012 陈劲安

总体架构

前端部分包括词法分析及语法分析，词法分析将输入字符串的关键词分析成 Token 序列，语法分析则根据递归下降程序构建抽象语法树，语法树构成后，再遍历语法树检查语义错误，若无错误再将 Ast 交给后端处理。

后端部分主要生成中间代码和目标代码，本编译器使用四元式作为中间代码，目标代码则由中间代码产生的中间指令进行翻译并输出 MIPS 汇编代码。

词法分析

词法分析部分在头文件 `LexicalAnalyzer.cpp` 中的 `initialTokens` 插入了关键词，并在 `Symbol.h` 列入了 Token ，主要分析都在 `LexicalAnalyzer` 上进行，分析过程是使用 `ptr` 和 `nextChar` 遍历整个代码并生成对应的 `Symbol` ，文法当中若需要读入多个字符的情况采用提前读的策略，生成的 `symbol` 则交给语法分析器进行分析。

语法分析

语法分析使用递归下降方法自顶向下分析，将每个非终结符实现成一个函数，由于文法是左递归文法，因此将文法改为 BNF 文法才有利于自顶向下分析，并在必要时使用提前读策略。每将一个 Ast 节点分析结束，则在分析函数返回该语句节点，例如 `Dec、BinaryExp、ReturnStmt、Func、Block` 等，从而可将各个语法分析过程中同时构造 Ast 。

错误处理

由于错误类型非常多，因此只实现课程组要求的错误。语法错误都会在语法上检测，而语义错误都会在 Ast 构造好后进行检测，并将每个错误都传入到 `ErrorHandling` 类里分析错误类型。

在分析错误处理时会使用符号表检查 LVal 和 CallExp 是否声明过，将每个变量和函数声明都 push 到符号表，并在退出 Block 后将该层的符号都弹出。变量符号表的结构为 `vector<map<string, Decl*>>` ，函数符号表结构为 `vector<map<string, Func*>>` 。

中间代码生成

中间代码是通过遍历 Ast 生成，参考了课程组给的中间代码设计，并再自行设计了四元式的中间代码，中间代码主要有 `IrFunc、BasicBlock` 和 `Inst` 结构组成，并使用侵入式链表数据结构将 `BasicBlock` 和 `Inst` 储存起来以便优化。在每个 `BasicBlock` 中存储着自己的 `label_id` 和 `Inst` 中存储着对应的 `Variable` 。

目标代码生成

IR 生成结束后，再由 `IR-MIPS` 翻译器 `Generator` 将每个 `IrFunc、BasicBlock、Inst` 进行翻译成汇编代码，而翻译过程只扫描一遍 `IR` 序列。翻译过程中实现了 `loadVar` , `assign`, `loadAddr` 等常用的操作封装成函数，便于翻译过程。

代码优化

本编译器主要围绕在算术优化，由于除法所带来的开销很大，因此根据论文 Division by Invariant Integers using Multiplication 的算法对除法指令进行优化，将除以常数的指令都以乘法和右移符号代替。乘法也根据乘以 2 次幂的情况下生成为左移指令。

在 `BinaryInst` 中若两个指令为常数，则直接在编译器进行运算，而不需要再生成多余的运算指令。

本编译器也将所有的 `const` 常量直接翻译成数字，因此运算时不需要再从栈内取出，而是可以直接通过立即数的指令进行运算。

总体实现技巧

实现编译器是一个大规模的项目，实现过程中需要仔细思考编译器的架构和接口，利用面向对象程序设计思路进行设计，通过抽象和封装减少模块的耦合性。

由于编译器模块繁多，需要在实现后进行大量的测试才能确保编译器运行的正确性。在实现过程中也需要遵循工程化的方法，进行多次的调试找出 bug 来源，而每实现/修复一个功能都会进行 git commit 以便记录实现过程，也方便自己查看是否还有类似的问题。在遍历 Ast 时同时输出对应的调试数据确保编译器运行过程是正确的。并充分利用辅助测试库的数据进行整合更利于我找出 bug 的来源，也经常使用 gcc 编译 testfile 检测输出的正确性。