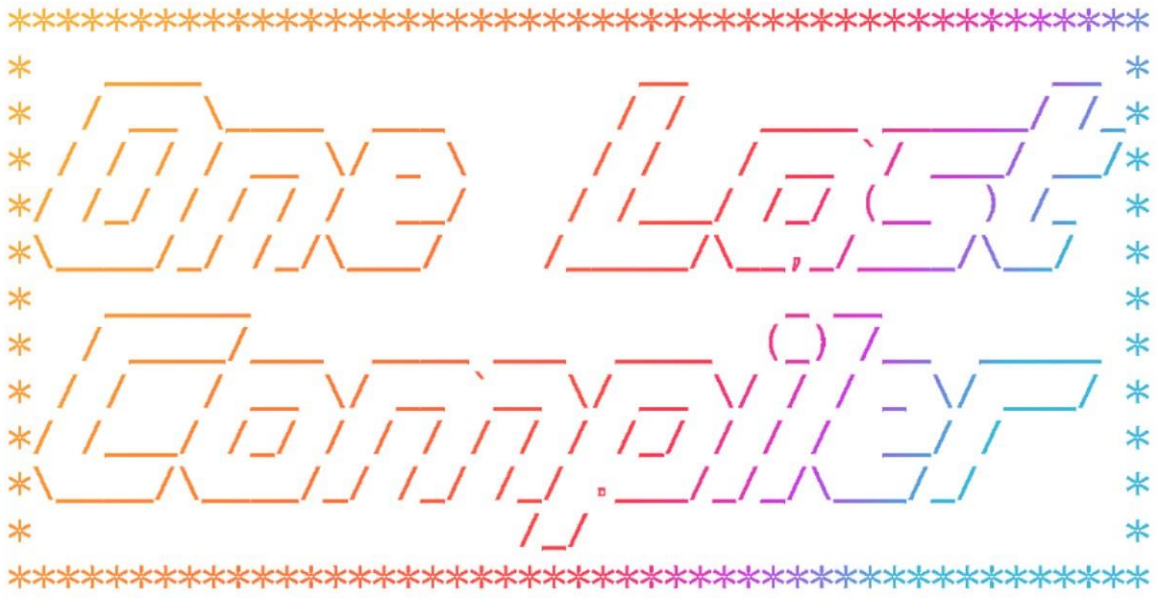
OneLastCompiler 设计文档



1. 项目概述

OneLastCompiler 是为 2024 年编译系统设计赛而开发的编译器。编译器支持 SysY2022 语言，这是一种简化的 C 语言子集，语言特性包括基本的数据类型、控制流语句、函数和数组操作等。编译器将 SysY2022 源代码(.sy 文件)编译成 ARMv7 指令集的汇编代码(.s 文件)，并且能够优化生成的代码，以充分利用 ARM 架构的特点，提高程序的运行效率。

2. 系统架构

项目使用语言为 C++17，开发时基于 CMake 构建。项目大体上分为前端和后端。前端使用 ANTLR 工具生成 C++的词法和语法解析器，并利用相应的 visitor 类，通过访问者模式，遍历输入文件的语法分析树(ParseTree)，进行语义分析，同时生成 SSA 形式的、类 LLVM 的中间代码(IR)。后端通过遍历中间代码 IR，最终生成 ARMv7 指令集的目标汇编代码。此外，在 IR 层面也将作出相应的优化 Pass，以提高生成的汇编代码的执行效率。

3. 详细设计

OneLastCompiler 项目的目标是构建一个高效、可扩展的编译器，以支持 SysY2022 语言并生成适用于 ARMv7 架构的汇编代码。编译器的架构设计关键在于清晰地划分各个处理阶段，以确保源代码能够高效转换成目标代码。以下是编译器的主要架构组成部分及其详细设计：

(1) 前端(frontend)

前端负责解析源代码，检查语法错误，并构建语法分析树(ParseTree)。这一阶段包括以下关键子系统：

- **词法分析器(Lexer)**：将源代码字符串转换成一系列标记(tokens)。这些标记是构建语法树的基本元素，包括关键字、运算符、标识符等。
- **语法分析器(Parser)**：解析标记序列，检查源程序的语法结构是否符合 SysY2022 的语法规则，并构建 AST。错误处理机制将在此阶段捕获并报告语法错误。

OneLastCompiler 使用 ANTLR 作为词法分析器和语法分析器的生成工具，以提高开发效率和代码质量。ANTLR 是一种强大的跨语言语法解析器，可以通过语法规则(.g4 文件)生成目标语言(如 C++)的词法解析器(Lexer)和语法解析器(Parser)，并能提供相应的 visitor 类以便通过访问者模式遍历语法分析树(ParseTree)。

使用 ANTLR 工具时，需提供.g4 语法文件。因此在开发时，首先要为 SysY2022 编写相应的语法文件 sysy2022.g4，然后再确保电脑上有 java11 运行环境的前提下，运行 ANTLR，生成若干.h 和.cpp 文件，包括词法分析器 Lexer 类、语法分析器 Parser 类、访问者类。其中，sysy2022Visitor.h 是访问者基类，sysy2022BaseVisitor.h 是该访问者基类的一个空的默认实现。

使用访问者模式遍历语法分析树时，只要继承自 sysy2022BaseVisitor 类，然后确定并编写各个语法树节点需要完成的语义动作即可。

ANTLR 需要运行时库，本地构建时将通过 CMake 相关配置自动拉取 ANTLR 运行时库代码。

(2) 中间表示(IR)

语法树 ParseTree 经过语义分析后，将转换成一种更接近机器语言的中间表示(IR)。IR 提供了一个与具体机器无关的代码表示方式，便于实现代码优化和目标代码生成。我们所采用的 IR 是类 LLVM 的、SSA 形式的 IR。

- **IR 生成**：将语法树 ParseTree 转换为我们的 IR，为接下来的优化阶段提供一个更加规范和易于操作的数据结构。
- **Pass**：在 IR 层面进行各种优化处理，如死代码消除、循环优化等，以提高代码效率和减少最终生成的代码量。

由于语法分析树 ParseTree 已经包含了所有的上下文信息，基本上类似于抽象语法树 AST，所以我们选择直接从语法分析树 ParseTree 生成中间表示 IR。在构建 IR 时，通过 CodeGenASTVisitor 类遍历该语法分析树，递归生成 IR 即可。此外，为了实现常量折叠(ConstantFolding)，我们使用另一个 visitor 类 ConstFoldVisitor，来处理常量折叠。总体上，我们通过一次遍历生成了中间表示 IR。

从前端到 IR 生成的其它一些重要的工作还包括符号表、类型系统、IR 相关类的继承等等，在

此不再赘述。

(3) 后端(backend)

后端负责将优化后的 IR 转换成目标机器 ARMv7 的汇编代码。这一过程涉及到多个重要的子阶段：

- **指令选择**：将 IR 指令转换为特定架构 ARMv7 的机器指令。
- **寄存器分配**：分析程序的变量使用情况，将变量分配到寄存器中。由于寄存器数量有限，这一步骤还包括必要的溢出处理，即将一些变量存储在内存中。
- **代码输出**：将最终的汇编代码输出到目标文件(.s 文件)。

其它一些重要的内容包括 GNU 汇编程序语法、ARM 函数调用约定、硬件浮点指令、ABI 函数的调用、运行时库的链接等等，不再赘述。

(4) 工具和环境支持

- **开发环境**：

  使用 CMake 进行构建，减少项目编译上的问题。

  在前端部分，使用 ANTLR 工具来生成词法分析器(Lexer)和语法分析器(Parser)，确保源代码的快速解析。
- **测试和调试**：编译器开发中包含广泛的单元测试和集成测试，以确保每个构建阶段的正确性和性能。我们的 IR 有相应的类似与 LLVM 的文本打印的形式，便于及时检查调试。最终测试时，使用 arm-linux-gnueabihf-gcc 交叉编译器+QEMU 模拟器，模拟指定硬件的运行，得到测试结果。

4. 未来计划

- **增加优化 Pass**：根据性能分析结果，增加更多的优化 Pass。