

LameCC

LameCC: 编译原理课程设计项目汇报

2050250 李其桐

Computer Science and Technology Department, College of Electronic and
Information Engineering(CEIE), Tongji University.
同济大学 电子与信息工程学院 计算机科学与技术系

2023 年 6 月 25 日

项目开发任务目标

- 类 C 编译器程序架构设计与实现：使用高级程序语言作为实现语言，设计并实现一个类 C 语言的编译器，编码实现编译器的组成部分。
- 词法分析任务：对于词法分析任务，给出类 C 语言的单词子集及机内表示，输入为源程序字符串，输出为单词的机内表示序列。
- 语法分析任务：对于语法分析任务，通过 LR(1) 或者递归下降等语法分析方法设计并且构建语法分析器，同时在语法分析过程中一遍地调用词法分析器的 nextToken 方法获取下一个 token，推进语法分析过程。
- 中间代码生成任务：使用语法制导翻译技术，选择合适的中间代码表示形式（本项目中采用 LLVM IR），要求能够在语法分析的同时生成中间代码，并且将生成结果保存到文件中。
- 目标代码输出任务：编译器能够根据输入类 C 语言源程序，还有运行时的参数选择，针对不同的处理器架构（target），输出例如 x86，x86-64，mips 等多种汇编代码，并且根据运行时参数进行不同等级的代码优化。生成后的汇编代码文件可以链接第三方编译环境提供的相关类库，进一步生成可执行文件。

项目开发任务目标

- 实现过程、函数调用、指针、数组和 GCC 风格内联汇编的代码编译：本实验中，我已经实现了包括过程、函数调用（支持递归）、指针、数组等各种类 C 语言的文法扩展和编译能力，使得编译器的功能更加健全与完备，具体实现的扩展功能点将在报告下文中进行详细阐述。

预备知识

- 词法分析器设计原理
- 文法分析方法，包括 LR(1) 分析方法、递归下降分析法，ACTION 表和 GOTO 表的推导方法以及分析流程逻辑等等
- 类 C 语言语法规范文法的设计
- 语义分析与中间代码产生原理与技术
- LLVM IR Builder 库编译、集成与部署技术，用以发射规范的 LLVM IR 中间代码，这种中间代码表示形式相比于四元式而言可读性和规范性更强，并且更加容易进行代码优化。
- 中间代码优化与目标代码生成技术

开发环境

- OS: Windows 11 Pro/Mac OS/Ubuntu20.04(可跨平台)
- Language: CPP
- IDE: vscode + visual studio
- 编译环境: Cmake + MinGW64 + MSVC + Clang
- 报告绘图工具: StarUML + Doxygen
- LLVM 版本: version 17.0.0 git Optimized build.

依赖库

- json.hpp (<https://github.com/nlohmann/json>): 用于以 json 格式 dump tokens
- rang.hpp (<https://github.com/agauniyal/rang>): 用于修改控制台输出字体的颜色、样式等。
- LLVM version 17.0.0 (<https://github.com/llvm/llvm-project>): 使用 `llvm::cl::opt` 处理程序运行时的启动参数, 使用 LLVM 提供的 LLVM IR Builder 相关接口生成 LLVM 中间代码, 使用 LLVM Pass 进行代码优化和目标代码生成。

核心算法设计

- 词法分析器：基本架构采用一个 DFA，由 nextChar() 获取的下一个字符标识 DFA 状态，数据粒度为字符，对外的接口是 run 和 nextToken(一遍过程中由 Parser 调用)
- 语法分析器：两套语法分析器分别采用 LR(1) 分析方法和递归下降语法分析方法。在实际开发中，随着我期望实现的语言特性越来越多，LR(1) 方法中的文法设计变得愈发困难，因此该方法实现的语法分析器在项目后期我已经停止维护，而使用递归下降的方法实现了所有下述的编译器功能特性。
- 中间代码生成器：实际开发中采用访问者模式实现，同样使用两套生成器，采用语法制导的翻译方法。对于代码中每个文法节点的 gen 方法传入不同的目标代码生成器对象指针，就能够生成不同表示形式的中间代码。
- 目标代码生成器：通过 LLVM Target 中提供的相关接口，直接将 LLVM IR 形式的中间代码映射到对应 target 下的目标代码，并且使用 LLVM Pass 实现不同等级的代码优化逻辑。
- 可执行文件生成：依赖第三方编译环境提供的类库实现和链接器

类设计

- **File 类**：用于储存读入的源文件内容，提供了操作文件的相关接口，包括数据获取、移动行号、信息记录等等
- **Lexer 类**：词法分析器实现主体类，在一遍过程中为语法分析器提供 nextToken() 接口用于推进分析过程
- **Parser 和 LR1Parser 类**：实现了递归下降和 LR1 分析两种语法分析方法的语法分析器实现主体类，输出相同的 AST 数据结构
- **LLVMIRGenerator 和 IRGenerator 类**：分别使用语法制导翻译技术，输出 LLVM IR 和四元式形式表示的中间代码序列
- **CodeGenerator 类**：使用 LLVM IR 中间代码序列作为输入，执行代码优化逻辑后输出可执行的目标文件
- **相关工具类**：包括错误处理、日志记录、运行时参数解析等于项目实现相关的工具类，可在源码中查看，此处不再赘述
- **AST 节点类**：本实验中的 AST 节点类设计参考了 Clang 源码中的设计架构，为每种实现需求中的文法符号实现了对应的节点类型定义，此处重点阐述其中所有类型的具体含义

AST 节点类型设计

- `lcc::AST::ASTNode`: 该类型是所有抽象语法树节点类型的基类（抽象类），作为一个接口类，声明了所有 AST 节点类型必须实现的 `asJson` 和 `gen` 方法
- `lcc::AST::Decl`: 所有声明性质节点的基类（抽象类）
- `lcc::AST::TranslationUnitDecl`: 该类型是 AST 的根节点类型，表示顶层的声明上下文，该节点的孩子是若干个顶层 Decl 节点
- `lcc::AST::NamedDecl`: 该抽象语法树节点表示可能具有名字的声明（或者定义）类型，本项目中的 `FunctionDecl`、`VarDecl` 以及其它具有名字的声明对应的 AST 节点类型均为该类型的派生类
- `lcc::AST::VarDecl`: 该抽象语法树节点表示变量的声明（或者定义），即类似 `int a;` 的声明语句或者 `int a = 1 + 2;` 的定义语句
- `lcc::AST::ParmVarDecl`: 该抽象语法树节点表示函数声明或者定义圆括号（paren）之间的形参，即类似... `...(int a, int b)` 中的 `int a, int b`
- `lcc::AST::FunctionDecl`: 该抽象语法树节点表示函数的声明（或者定义），即类似 `int func(int param);` 的声明语句或者 `int func(int param) ...` 这样的定义语句

AST 节点类型设计

- `lcc::AST::Expr`: 所有表达式性质节点的基类（抽象类）
- `lcc::AST::BinaryOperator`: 该抽象语法树节点类型表示一个双目运算符表达式，维护表达式的左部，符号和右部信息（例如 `a+b`）
- `lcc::AST::UnaryOperator`: 该抽象语法树节点类型表示一个单目运算符表达式，维护表达式的符号和右部信息（例如 `!a`）
- `lcc::AST::CastExpr`: 该抽象语法树节点类型表示一个类型转换表达式，所有类型转换表达式类型的基类
- `lcc::AST::ImplicitCastExpr`: 该抽象语法树节点类型表示一个隐式类型转换表达式
- `lcc::AST::DeclRefExpr`: 该抽象语法树节点类型表示一个对变量或者函数的引用
- `lcc::AST::CallExpr`: 该抽象语法树节点类型表示一个函数调用表达式，即形如 `func(1,2)` 之类的式子
- `lcc::AST::ParenExpr`: 该抽象语法树节点类型表示一个括号表达式，其基本形式为 `(Expr)`，即前缀和后缀分别为左右括号 `(Expr)`，中间包裹一个表达式。

AST 节点类型设计

- `lcc::AST::IntegerLiteral`: 该抽象语法树节点类型表示一个整形数字面量, 即数值 1, -2,321 等等类似的整形数
- `lcc::AST::FloatingLiteral`: 该抽象语法树节点类型表示一个浮点数字面量, 支持下列 6 种浮点数表示法

AST 节点类型设计

- `lcc::AST::Stmt`: 该抽象语法树节点类型表示一个语句, Clang 中对于该类型的描述如下
- `lcc::AST::CompoundStmt`: 该抽象语法树节点类型表示一个混合语句。所谓混合语句, 就是以大括号围起来的若干语句集合
- `lcc::AST::WhileStmt`: 该抽象语法树节点类型表示一个 While 循环语句, 即类似 `while(1) ...` 的语句
- `lcc::AST::IfStmt`: 该抽象语法树节点类型表示一个 if 条件分支语句, 包括 if 和 if...else... 两大类。
- `lcc::AST::ReturnStmt`: 该抽象语法树节点类型表示一个返回语句, 包括返回一个值 (即表达式) 或者返回空两种情况
- `lcc::AST::NullStmt`: 该抽象语法树节点类型表示一个空语句
- `lcc::AST::DeclStmt`: 该抽象语法树节点类型表示一个变量声明或者定义语句
- `lcc::AST::ValueStmt`: 该抽象语法树节点类型表示数值语句, 即一个表达式 (Expr)
- `lcc::AST::AsmStmt`: 该抽象语法树节点类型表示一个 GCC 风格内联汇编语句, 语法规则如下图所示

AST 节点类型设计

```
asm asm-qualifiers ( AssemblerTemplate
                    : OutputOperands
                    [ : InputOperands
                    [ : Clobbers ] ] )
```

```
asm asm-qualifiers ( AssemblerTemplate
                    : OutputOperands
                    : InputOperands
                    : Clobbers
                    : GotoLabels)
```

AST 节点类型设计

- `lcc::AST::ArraySubscriptExpr`: 该抽象语法树节点类型表示一个数组索引语句（比如 `a[0]`, `arr[idx]` 等等）

增加单词的数量

编译器支持的所有关键字在 `TokenType.inc` 中定义，可进入源码查看



将整常数扩充为实常数

本次项目开发中实现的词法分析器已经能够支持按照 C 语言语法标准读入实常数，包括整形数和浮点数。此处给出所有接受的浮点表示形式如下：

```
float f1 = 1.1;
float f2 = 1.2f;
float f3 = 9.2E+3; // scientific notation with positive exponent
float f4 = 8.4E-2; // scientific notation with negative exponent
float f5 = 0xAF.EP+4; // p-notation with positive exponent
float f6 = 0xAF.D65P-5; // p-notation with negative exponent
```

增强编译器的编译能力

编译器在基础项目开发要求的基础上扩展了编译能力，支持包括数组、指针、GCC 风格内联汇编语句等等一系列基本 C 语言语法规则，具体支持的所有 C 语言语句可以到源码下的 test.c 测试源码文件中查看

较为完善的错误处理

对于输入的源码文件，编译器能够给出存在词法错误的 token（包括错误类型和位置）还有存在语法错误的语句（包括错误类型和位置），便于开发人员定位错误并且修改



从词法分析到可执行文件生成的全套解决方案

本编译器最终生成的目标代码文件（例如 x86-64 汇编或者 x86 汇编），可以直接输入到对应的第三方编译环境中（例如 GCC, Clang 等），进一步通过第三方编译环境提供的链接器与第三方实现的标准类库进行链接，最终生成能够在对应平台上正确执行的可执行文件，从而更加直观地查看编译器对于源码逻辑编译的正确性

项目源码地址

<https://github.com/leo4048111/LameCC>



总结

汇报结束，请老师批评指正！

