重庆大学编译原理课程实验报告

实验题目	编译器设计与实现

一、实验目的

以系统能力提升为目标，通过实验逐步构建一个将类 C 语言翻译至汇编的编译器，最终生成的汇编代码通过 GCC 的汇编器转化为二进制可执行文件，并在物理机或模拟器上运行。实验内容还包含编译优化部分，帮助深入理解计算机体系结构、掌握性能调优技巧，并培养系统级思维和优化能力。

二、实验项目内容

本次实验将实现一个由 SysY (精简版 C 语言，来自 https://compiler.educg.net/) 翻译至 RISC-V 汇编的编译器，生成的汇编通过 GCC 的汇编器翻译至二进制，最终运行在模拟器 qemu-riscv 上

实验至少包含四个部分: 词法和语法分析、语义分析和中间代码生成、以及目标代码生成，每个部分都依赖前一个部分的结果，逐步构建一个完整编译器

实验一：词法分析和语法分析，将读取源文件中代码并进行分析，输出一颗语法树

实验二：接受一颗语法树，进行语义分析、中间代码生成，输出中间表示 IR (Intermediate Representation)

实验三：根据 IR 翻译成为汇编

实验四(可选)：IR 和汇编层面的优化

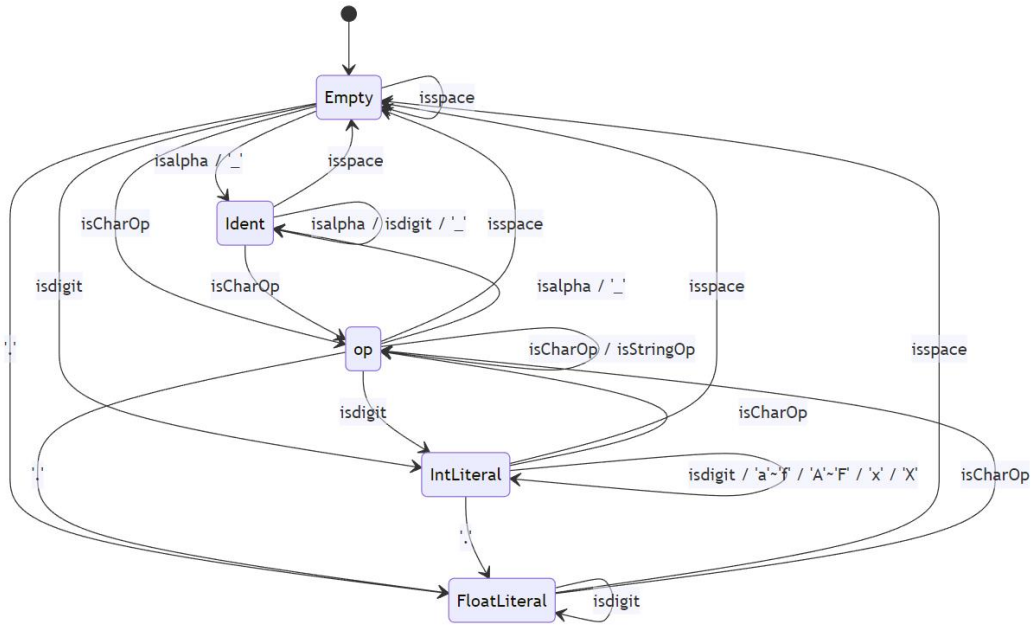
三、实验内容实现

1. 实现了哪些内容

· **实验一（47/47）**：实现编译器前端的词法分析和语法分析部分，分析输入的源文件得到一颗抽象语法树，具体实现如下：

词法分析：使用有限状态机模型，在不同状态下处理对应的 Token 并转移状态，使用 cur_str 和 cur_state 分别记录当前字符串和状态，将作业 1 中主函数构造 token 数组的过程封装为 Scanner::run()函数，返回 token 数组。词法分析阶段只管分词，不考虑语法问题。因为 INTLTR 型分词只有在接收到非 INTLTR 型输入时才能够跳出 IntLiteral

状态，所以所有分词获取成功后都在下一状态输出。返回 true 表示即将输出 buf，只有返回 true 前才更新 buf.value 和 buf.type。为了最后一个分词能够输出，在 stdin_str 末尾加上了 "\n"。状态转移如下：



返回 true 前需要将 cur_str 赋给 buf.value，并调用 get_op_type 函数赋值 buf.type，然后调用 reset 函数。返回 true 时接收的输入（即当时的 input）被保存在 cur_str 中，不会被丢弃。

语法分析：由于文法规则加入了许多需要递归的产生式，因此作业 2 的思路不再适用。定义辅助函数 Match_XX 为每种产生式的构造提供可选的终结符/非终结符，以集合方式返回。定义辅助函数 retreat 为匹配失败的情形恢复原状态。使用递归下降法生成抽象语法树，在构造产生式时从左向右依次判断当前 token 是否符合文法规则，符合则匹配成功，失败则回退。构造产生式的函数除了 CompUnit 返回 AstNode 类型，其余均为 bool 型。需要注意的是，UnaryExp -> PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp 文法因为 PrimaryExp 分支可直接由 Ident 构成，与 Ident '('分支冲突，若直接使用回溯，遇到单个 Ident 就会生成 PrimaryExp 成功，无法回溯。因此不能单纯使用回溯实现，而是需要手动实现 LL(2)分析（该函数处理思路借鉴 https://gitee.com/wangxingran222/Compiler_Theory_EXP1/blob/master/src/front/syntax.cpp）。

- **实验二（58/58）：**由于本地环境不兼容，从实验二开始使用 WSL/Ubuntu + Docker 环境进行编译。具体实现如下：

语义分析的核心函数是 get_ir_program，使用语法指导翻译技术，对实验 1 得到的抽象语法树 AST 使用 DFS 分析源程序，得到中间表示 IR。在 semantic.cpp 中，为每条文法规则定义 analyze_XX 函数以实现该文法的翻译，翻译时根据子节点的类型和属性计算父节点的类型和属性，并将遇到的符号压入符号表数组，以及按要求返回 IR 指令的数组（如果有）。实验中节点类型有 Int、Float、IntLiteral、FloatLiteral，在计算表达式时需要隐式类型转换，因此定义了 IntLiteral2Int()、IntLiteral2FloatLiteral()、IntLiteral2Float()、Int2Float()、FloatLiteral2Float()函数进行变量转换。

由于实验框架中定义的作用域结构体允许嵌套且不同作用域中可以定义同名变量，因此在翻译成 IR 的过程中需要为不同作用域中的同名变量重命名，本实验采用的重命名法是作用域+变量名。需要注意的是，逻辑运算表达式具有短路特性，翻译时不能不加判断就将所有逻辑全部翻译。还有一个是群里经常有同学问的声明全局数组问题，不需要手动 alloc 数组，否则会让被初始化为随机值值而不是 0。

- **实验三（54/58）：**根据实验二生成的 IR 程序数组生成 risc-v 汇编，需要先解析全局变量，生成全局变量的初始化汇编，然后按顺序解析 program.functions 中的所有函数，根据每个函数的 IR 指令生成对应的 risc-v 汇编指令。由于 risc-v 汇编需要在开头引入启动代码，因此定义 set_nopic()函数和 set_text()函数以生成基本头部的汇编。

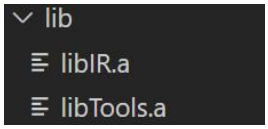
risc-v 汇编中比较重要的就是栈和寄存器的分配。实验中分配栈时，需要事前计算其大小，即在生成每个函数的汇编前先遍历一遍函数的 IR 指令，根据局部变量的数量分配栈的大小。但这些变量都存在内存中，只有在变量需要被使用时才为其分配寄存器，然后从内存中载入寄存器，使用结束后就存回内存，收回寄存器。

由于最后考试周时间有限，没能实现浮点数的 IR 到 risv-v 汇编的过程，实验三最后只通过了 54/58 个测试点。

2. IR 库的使用，如何使用静态库链接，如何使用源代码来构建库？结合 CMakelist 说明

静态库链接就是将静态库中的目标代码（即被应用程序引用的所有函数和数据）直接链接（即复制）到目标程序中的过程。一个静态库包含了多个目标文件，它们被打包到一起以便于分发和链接，这些目标文件通常是用 C 或 C++编写的源代码经过编译而成的。在链接阶段，链接器将查找程序代码中引用的所有符号，并尝试在静态库中找到这些符号的定义，如果找到，那么这些符号的代码或数据就会被直接链接到最终的可执行文件中。本实验使用 lib 文件夹下的静态库 libIR.a 和 libTools.a，在 CMakeLists.txt 文件

中添加 link 指令来链接静态库：




```
# -------------------- from lib --------------------
# link libxx.a
# u should rename libxx-x86-win.a or libxx-x86-linux.a to libxx.a according to ur own platform
link_directories(./lib)
# -------------------- from lib --------------------
```

使用静态库链接方式的优点是生成的可执行文件是自包含的，它不依赖于任何外部的库文件，因此分发和执行都相对简单。缺点是静态链接可能会导致程序的大小增大，并且无法利用动态链接库带来的某些好处，比如运行时共享代码或在不需要重新编译程序的情况下更新库函数。

使用源代码来构建库则是将指定文件夹下的源代码文件编译成库文件，可以是静态库，也可以是动态库。这个过程发生在编译阶段，编译器将源代码文件编译成目标文件，然后这些目标文件被打包成一个库文件。这样就可以在多个程序中重用这些代码，而不是每次都将这些代码编译到程序中。构建库需要在 CMakeLists.txt 文件中添加 add_library 指令：

```
# build library
# every src file directory should be compile into a lib

aux_source_directory(./src/front FRONT_SRC)
add_library(Front ${FRONT_SRC})

# 为了 debug 方便，你可以选择通过源文件来构建 IR 测评机，但是请以链接静态库文件的方式去跑分
# -------------------- from src --------------------
aux_source_directory(./src/ir IR_SRC)
add_library(IR ${IR_SRC})
aux_source_directory(./src/tools TOOLS_SRC)
add_library(Tools ${TOOLS_SRC})
# -------------------- from src --------------------
```

3. 在 IR 中你如何处理全局变量的，这样的设计在后端有什么好处？后端中如何处理全局变量？

处理全局变量集中在实验二 IR 生成的过程中，我将所有全局变量都存在 Analyzer 的全局符号表 symbol_table 中。在核心函数 get_ir_program 中，第一步就是向全局符号表的作用域数组中压入作用域，该作用域就是全局作用域。添加全局作用域后，DFS 分析语法树构建 IR，构建完成后单独处理全局变量，即对 symbol_table.scope_stack[0].table

中的元素（也就是全局变量）进行处理。处理过程使用 process_global_var 函数，通过判断变量类型将其压入 ir::Program.globalVal 中进行存储：

```
void frontend::Analyzer::process_global_var(frontend::STE &entry){
    auto &operand = entry.operand;
    std::string scoped_name = symbol_table.get_scoped_name(operand.name);
    Type type = operand.type;

    // 该条目是数组
    if (entry.dimension.size() != 0) {
        int arr_len = 1;
        for (auto dim : entry.dimension) {
            arr_len *= dim;
        }
        ir_program.globalVal.push_back({{symbol_table.get_scoped_name(entry.operand.name),entry.operand.type}, arr_len});
    } else {
        // 该条目是标量
        switch (entry.operand.type) {
            case Type::FloatLiteral:
                ir_program.globalVal.push_back({{symbol_table.get_scoped_name(entry.operand.name),Type::Float}});
                break;
            case Type::IntLiteral:
                ir_program.globalVal.push_back({{symbol_table.get_scoped_name(entry.operand.name),Type::Int}});
                break;
            default:
                ir_program.globalVal.push_back({{symbol_table.get_scoped_name(entry.operand.name),entry.operand.type}});
                break;
        }
    }
}
```

这样的设计便于后端对全局变量的统一处理，即后端只需解析全局函数和全局变量表即可处理所有全局变量。在后端中，我定义了 solve_global_data()函数专门对全局变量声明、初始化数组和未初始化数组进行处理。

4. 在函数调用的过程中，汇编需要如何实现，汇编层次下是怎么控制参数传递的？是怎么操作栈指针的？

在函数调用前，调用者（即 Caller）保存所有它需要在函数调用时使用的寄存器的原值，这些寄存器的值被保存到栈中，以便在函数返回时将这些寄存器恢复原值。在 risc-v 架构中，前 8 个函数参数被放置在 a0 到 a7 寄存器中，如果参数超过八个，额外的参数需要保存至栈中。调用函数时执行 jal 指令，此时程序计数器 PC 跳转到函数的入口，同时将返回地址存储在 ra 寄存器中以便函数调用结束后返回原 PC。被调用函数（即 Callee）先要加载函数参数，小于等于 8 个的参数从 a0~a7 读，多余部分从 Caller 的栈里面取，即去当前函数的栈顶指针+(i-8)*4 内存地址处依次取。当 Callee 执行完成后就会返回，返回值会保存 a0 寄存器中，函数跳转到 ra 寄存器中的地址返回。在返回之前，Callee 还需要恢复所有在函数开始时保存的寄存器的值。

四、实验测试

编译的前提是拥有完整且正确的环境以及符合要求的 CMakeList.txt，配置完成后才

可以进行测试。我的配置过程如下，WSL/Ubuntu+Docker 配置是配置 docker 时写的博客：

环境配置：

由于本地环境不兼容，因此本实验采用 WSL/Ubuntu + Docker 环境进行编译，配置过程参考 WSL/Ubuntu+Docker配置。配置完成后按照指导书上步骤拉取镜像、指定编译器、创建容器、挂载目录：

1. 拉取镜像：`docker pull frankd35/demo:v3`；
2. 在 CMakeLists.txt 中指定 x86-linux 的编译器：

```
# 增加以下指令以兼容 Docker
set(CMAKE_C_COMPILER    "/usr/bin/x86_64-linux-gnu-gcc-7")
set(CMAKE_CXX_COMPILER  "/usr/bin/x86_64-linux-gnu-g++-7")
```

3. 在 CMakeLists.txt 中修改所需 CMake 版本

```
# 修改以下指令以兼容 Docker
cmake_minimum_required(VERSION 3.10)
project(compiler)
```

4. 创建容器并挂载目录：`docker run -it -v /home/scienceli1125/Project/Lab2:/coursegrader frankd35/demo:v3`。因为我的代码框架目录被放在 \wsl.localhost\Ubuntu-22.04\home\scienceli1125\Project\Lab2 下，所以需要将其映射到 Docker 容器的 coursegrader 下。执行该指令后，Docker 会创建并运行一个新的容器并为该容器分配一个唯一的容器 ID，界面进入以 root 身份登录的容器命令行，那长串字符串就是容器的 ID。如果想要结束当前容器，执行 `exit` 即可。挂载目录可以使容器能够访问和操作 Linux 上的文件，而无需将这些文件复制到容器内部；

配置完环境并且编写完代码后，就可以编译执行了，以下是使用单独样例文件进行测试并输出结果的流程：

进入容器：

1. `sudo service docker start`：启动 Docker；
2. `docker start 204996ca5ffd`：运行容器；
3. `docker exec -it 204996ca5ffd bash`：进入容器；
4. `cd coursegrader`：进入挂载目录；

编译：

1. `cd build`：进入 build 文件夹；
2. `cmake ..`：读取 CMakeLists.txt 文件获取项目的构建配置和规则，生成构建配置和相应的构建脚本；
3. `make`：根据 Makefile 中的规则和依赖关系，自动化地编译源代码并生成可执行文件；

执行：

1. `cd ../bin`：进入 bin 文件夹；
2. `./compiler <src_filename> [-step] -o <output_filename> -O1`：使用单独样例文件进行测试并输出结果；

   -step 支持以下几种输入：

   s0: 词法结果 token 串

   s1: 语法分析结果语法树, 以 json 格式输出

   s2: 语义分析结果, 以 IR 程序形式输出

   S: RISC-v 汇编

   <src_filename>: 测试样例文件路径

   <output_filename>: 输出样例文件路径

   如: `./compiler ../test/testcase/basic/00_main.sy S -o test.out -O1`

上述两个步骤适用于编写与调试阶段，支持单个文件测试以及连接 gdb 调试。如果程序已经无误，可以使用 python 脚本进行测试。

单个文件样例测试完成后，可以直接运行 test 文件夹下的 python 脚本，使用

CMakelist.txt 中配置的参数和 build 指令执行编译 compiler，然后 run 指令执行编译好的二进制 compiler 程序，把每个输入对应的输出结果输出到 output 文件夹下，最后执行 score 指令调用 diff 程序将 output 文件夹下的输出与标准输出内容对比以判断该测试点是否通过：

测试:

1. `cd ../test`：进入 test 文件夹;
2. `python3 build.py`：进入到 build 目录, 执行 `cmake ..` 和 `make` 语句;
3. `python3 run.py S`：运行可执行文件 compiler 编译所有测试用例, 打印 compiler 返回值和报错, 输出编译结果至 /test/output。注意使用 python3 执行，因为 subprocess.run 是在 Python 3.5 版本中引入的新函数，旧版 Python 无法执行;
4. `python3 score.py S`：将 run.py 生成的编译结果与标准结果进行对比并打分;

汇编变成 RISV 程序被执行的过程如下：实验三生成的汇编文本文件被汇编器 riscv32-linux-gnu-gcc 转换为机器语言指令文件，然后与静态库链接，生成 risc-v32 二进制文件，最后在 RISC-V 架构模拟器 qemu 上调用该可执行文件，模拟 risc-v32 在硬件上的执行过程，生成结果。

五、实验总结

1. 实验过程中所遇到的问题及解决办法

• 问题 1：词法分析时获取 INTLTR 型 Token 时遗漏数字，如 100 只切分出 0。但输出过程中的 cur_str 并未发现异常。

解决办法：在 Switch 中加了很多断点以及输出中间变量都没有发现问题，于是怀疑是 Switch 前的补全 buf.type 语句造成的数字缺失：

```
if(cur_str=="+" || cur_str=="-" || cur_str=="*" || cur_str=="/" || cur_str=="(" || cur_st
    cur_state = State::op;
    buf.type = get_op_type(cur_str);
}else if(cur_str == "0" || cur_str == "1" || cur_str == "2" || cur_str == "3" || cur_str
    cur_state = State::IntLiteral;
    buf.type = get_op_type(cur_str);
}else{
    cur_state = State::Empty;
}
```

该段代码是为了识别 INTLTR 成功后记录当前 input 并在下一次调用前补全该字符的 buf.type，于是将其注释后选取含有空格的样例（空格不担心被吞掉），发现能够成功识别。在返回 false 的情况下，只需要进行 cur_str 和 cur_state 的变更，不需要更新 buf.value 和 buf.type，只有在返回 true（即输出前）的情况下才需要更新 buf.value 和 buf.type。因此重新设计 get_op_type 函数并修改 Switch 的逻辑，修改后 AC。

• 问题 2：语法分析时编写 Parser 内置函数 OP_Number() 时，先判断八进制后判断二进制导致二进制数被误判为八进制，错误输出：

```
if(val.length()>=3 && val[0]=='0' && (val[1]=='x' || val[1]=='X')){ // hexadecimal intege
    ...
}
else if(val.length()>=2 && val[0]=='0'){                          // octal integer
    ...
}
else if(val.length()>=3 && val[0]=='0' && (val[1]=='b' || val[1]=='B')){    // binary int
    ...
}
else{           // decimal integer (or float)
    ...
}
```

解决办法：改为先判断二进制后判断八进制即可：

```
if(val.length()>=3 && val[0]=='0' && (val[1]=='x' || val[1]=='X')){ // hexadecimal intege
    ...
}
else if(val.length()>=3 && val[0]=='0' && (val[1]=='b' || val[1]=='B')){    // binary int
    ...
}
else if(val.length()>=2 && val[0]=='0'){                          // octal integer
    ...
}
else{           // decimal integer (or float)
    ...
}
```

• 问题 3：因为内存中没有二维数组，也不知道如何对其进行标记，所以一开始生成 IR 时也没有单独处理二维数组，导致一些测试点 WA...

解决办法：条件判断时单独判断二维数组，并且在内存分配时使用普通数组的方式进行分配（如 int a[2][2] => alloc a, 4）即可。

• 问题 4：还有全局数组的初始化问题，一开始使用 alloc 手动分配内存，但会出现段错误或其他数组紊乱问题。

解决办法：添加 alloc 后 IR 执行机会使用系统调用分配内存，导致之前已经初始化的数组内存被换为未初始化的随机数据内存。经同学提醒，去掉全局数组的 alloc 后，错误解决。

2. 对实验的建议

虽然我也很想 RTFD，但奈何才疏学浅，指导书还是超出了我的理解与能力范围，以至于不得不向已经完成的大佬一次又一次地请教。这段历时 3 个月手撕编译器的经历让我再一次回想起几个月前扛着新冠对抗硬综却打死接不上 AXI 只能对着文档叹气的痛苦回忆，如芒在背，如鲠在喉。我知道教改是一个循序渐进且颇具难度的过程，我们不必抱怨，只想让后面的学弟学妹不再受我们受过的不必要的痛苦。希望下一届的实验指导书能够循序渐进，由浅入深，增加适当的实验引导，给出每个实验的大致思路。毕

竟对于陌生的实验框架，我们并不知道设计者设计这些结构体的目的和他想要使用的场景。同时建议理论课上加入一定的实操教学，而不是一味的讲述各种高级的编译技巧扩展，正如刘骥老师在最后一堂课上所说，编译原理的课堂应该是理论结合实践，而不是现在这样对着书本喋喋不休。