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		
实验时间	2024/6/22	实验地点	DS3 304
实验成绩		实验性质	□验证性 □设计性 ■综合性

教师评价：

□算法/实验过程正确；□源程序/实验内容提交 □程序结构/实验步骤合理；

□实验结果正确； □语法、语义正确； □报告规范；

其他：

评价教师签名：

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

0. 环境准备

实验基于 **WSL（Ubuntu 22.04） ＋Docker＋VScode** 环境。

1. 实现内容

1.1 实验一（47/47）

通过完善实验代码包中的 **lexical.cpp、syntax.h** 等代码文件，实现编译器前端的词法分析和句法分析，具体实现如下面所示：

1.1.1 词法分析

词法分析实验中，通过 **Scanner** 读入外部的 **C** 源程序中有效的代码部分，即提前去除注释掉的代码部分，并对其进行扫描，通过将字符输入到 **DFA** 中，**DFA** 把它们组成有意义的词素序列，对于每个词素，词法分析器都会产生词法单元 **token** 作为输出。在 **DFA** 中我们使用变量 **cur_state** 和 **cur_str** 来记录 DFA 当前的状态以及已经接受的字符串，通过输入进来的字符以及当前自身状态来决定转移后的状态，如果产生 **Token** 则返回。在本实验的 DFA 中设置了 **5** 中状态，即：

Empty：初始（空）状态，等待输入字符

Ident：标识符状态，即正在解析一个标识符或关键字

IntLiteral：整数字面量状态，即正在解析一个整数

FloatLiteral：浮点数字面量状态，即正在解析一个浮点数

op：操作符状态，即正在解析一个操作符。

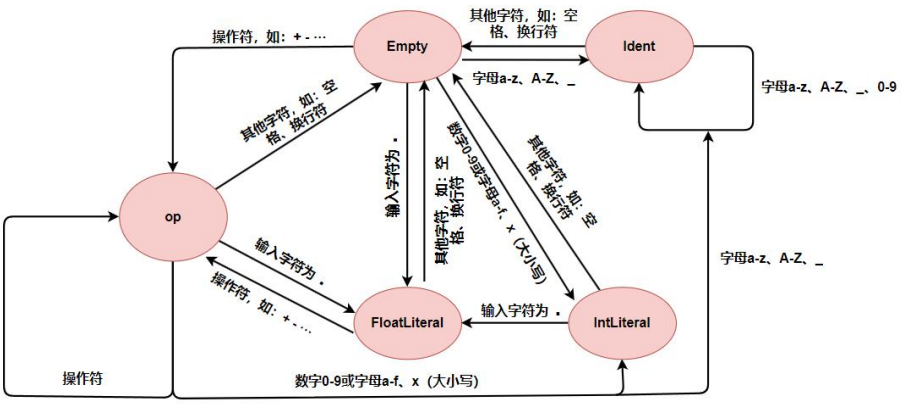
同时通过两个函数，即：

Combined_op(char c)：返回字符 c 可以结合的另一个字符，

如 **c＝‘＞’**，返回 **‘＝’**；如果不可以结合，则返回空。

isoperator(char c)：判断字符 c 是否是操作符。

辅助 **DFA** 进行状态转移和结果输出，下面是兼顾实验二的 **DFA 的**状态转移图（部分）：



1.1.2 语法分析

语法分析实验中，通过构建解析器 **Parser** 对词法分析产生的 **token** 数组采用递归下降法生成抽象语法树（**AST**），在具体递归下降法的实现中我们采用 **LL(1)**分析的方法，对每一条文法产生式创建对应的辅助解析函数，如图所示：

```
1  Term* parseTerm(AstNode* root, TokenType type); // 终结符
2  bool parseCompUnit(AstNode* root);
3  bool parseDecl(AstNode* root);
4  bool parseFuncDef(AstNode* root);
5  bool parseConstDecl(AstNode* root);
6  bool parseBType(AstNode* root);
7  bool parseConstDef(AstNode* root);
8  bool parseConstInitVal(AstNode* root);
9  bool parseVarDecl(AstNode* root);
10 bool parseVarDef(AstNode* root);
11 bool parseInitVal(AstNode* root);
12 bool parseFuncType(AstNode* root);
13 bool parseFuncFParam(AstNode* root);
14 bool parseFuncFParams(AstNode* root);
15 ...........
```

每个解析函数根据对应的文法产生式，每个函数根据自己的文法规则解析子节点或者解析终端节点，但是在构建辅助函数的时候我们只有终结符 token，所以我们要先求得每个非终结符的 first 集合（部分），如图：

```
FIRST(CompUnit) = FIRST(Decl) U FIRST(FuncDef) = {'conts', 'int', 'float', 'void'}
FIRST(Decl) = FIRST(ConstDecl) U FIRST(VarDecl) = {'conts', 'int', 'float'}
FIRST(ConstDecl) = {'const'}
FIRST(BType) = {'int', 'float'}
FIRST(ConstDef) = {Ident}
FIRST(ConstInitVal) = FIRST(ConstExp) U { '{' } = { '{', '(', Ident, IntConst, floatConst, '+', '-', '!' }
FIRST(VarDecl) = FIRST(BType) = { 'int', 'float' }
FIRST(VarDef) = {Ident}
FIRST(InitVal) = FIRST(Exp) U { '{' } = { '{', '(', Ident, IntConst, floatConst, '+', '-', '!' }
FIRST(FuncDef) = FIRST(FuncType) = { 'void', 'int', 'float' }
FIRST(FuncType) = { 'void', 'int', 'float' }
FIRST(FuncFParam) = FIRST(BType) = { 'int', 'float' }
FIRST(FuncFParams) = FIRST(FuncFParam) = { 'int', 'float' }
FIRST(Block) = { '{' };
```

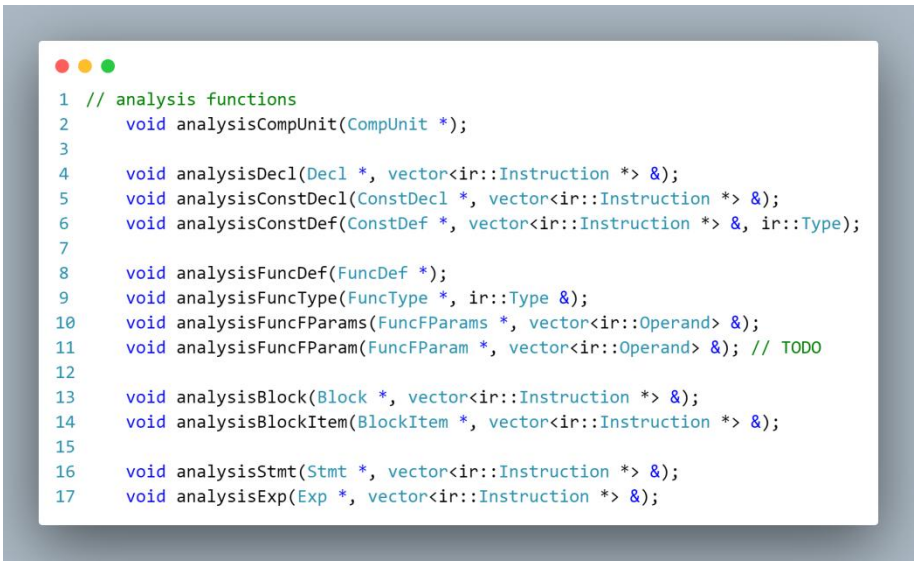
接下来构建辅助函数，以 **parseDecl** 为例：

```
bool  Parser::parseDecl(AstNode* root){
    /*
     * Decl -> ConstDecl | VarDecl
     * FIRST(ConstDecl) = { 'const' }
     * FIRST(VarDecl) = FIRST(BType) = { 'int', 'float' }
    */
    if(token_stream[index].type == TokenType::CONSTTK){  // const关键字
        PARSE(constdecl_node, ConstDecl);
        return true;
    }else if(token_stream[index].type == TokenType::INTTK || token_stream[index].type == TokenType::FLOATTK){
        PARSE(vardecl_node, VarDecl);
        return true;
    }else return false;
}
```

以最后形成一颗 **root** 为根节点的抽象语法树，在使用实验代码包提供的 **json** 工具可视化该语法树。

1.2 实验二（57/58）

该实验主要目的是实现语义分析过程以及语法制导翻译，生成中间表示 IR，通过完善实验包中的 **semantic.h** 以及 **semantic.cpp** 代码文件，以实验一语法分析的输出抽象语法树为输入，通过 Analyzer 进行分析和翻译，在结构上类似与语法分析，对每一条产生式构建一个解析函数结合指导书中 IR 的解释，进行语义分析和翻译，部分如图：



```
// analysis functions
    void analysisCompUnit(CompUnit *);

    void analysisDecl(Decl *, vector<ir::Instruction *> &);
    void analysisConstDecl(ConstDecl *, vector<ir::Instruction *> &);
    void analysisConstDef(ConstDef *, vector<ir::Instruction *> &, ir::Type);

    void analysisFuncDef(FuncDef *);
    void analysisFuncType(FuncType *, ir::Type &);
    void analysisFuncFParams(FuncFParams *, vector<ir::Operand> &);
    void analysisFuncFParam(FuncFParam *, vector<ir::Operand> &); // TODO

    void analysisBlock(Block *, vector<ir::Instruction *> &);
    void analysisBlockItem(BlockItem *, vector<ir::Instruction *> &);

    void analysisStmt(Stmt *, vector<ir::Instruction *> &);
    void analysisExp(Exp *, vector<ir::Instruction *> &);
```

在翻译过程中，要注意子节点和父节点的节点类型，以及不同作用域下的同名变量的区分等，同时穿插着变量类型的隐式转换，虽然做了一些挣扎但是最后还是没有实现浮点数计算。最后根据得到的 **IR** 指令链接 **IR** 执行器，得到 **c** 程序的输出和 **main** 函数返回值。

1.3 实验三（57/58）

通过完成实验的最后一个部分，我们将实验二生成的中间代码 IR 转换为可以与 **sylib.a** 链接的 **RISC-V** 汇编代码。实验的主要实现过程如下（由于实验二中实现了浮点数运算，实验三仅实现整型部分）：

- **全局变量解析**：包括整型变量和数组变量的声明与空间分配。确保每个全局变量在内存中都有正确的存储位置。
- **解析代码中的函数**：完成接口 **gen_func()**的实现。这包括处理函数的定义、参数的传递、局部变量的声明与初始化等。
- **生成 RISC-V 指令**：在生成 **RISC-V** 汇编代码时，需要注意栈空间以及寄存器的分配。我们采用计算后立即存储的方法，这意味着同时使用的临时寄存器不会超过三个。因此，寄存器分配方案采用“写死”策略，同时封装了一个简易接口用于申请和回收寄存器。
- **辅助函数的实现**：在生成 **store** 和 **load** 指令时，需要处理对全局变量和局部变量的访问。此外，在函数调用中涉及传值和引用传递时，还需要处理解引用的问题。为了简化这些操作，创建了 **store** 和 **load** 辅助函数，以便正确处理不同类型的内存访问。

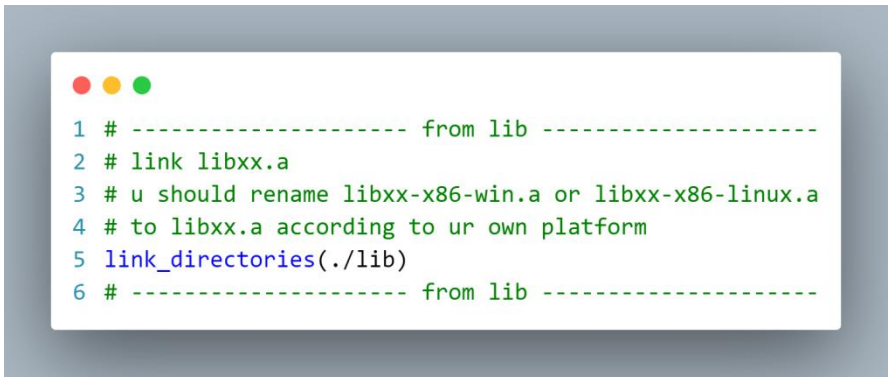
通过这些步骤，能够将中间代码成功转换为可链接的 **RISC-V** 汇编代码，并确保其在与 **sylib.a** 链接时能够正常运行。

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

2.1 使用静态链接库

通过链接预编译的静态库文件来构建项目，这种方式通常更快，因为库已经编译完成，只需要链接即可，但是不便于调试和修改。在 **CmakeLists.txt** 中通过如下代码直接链接静态库，如图：

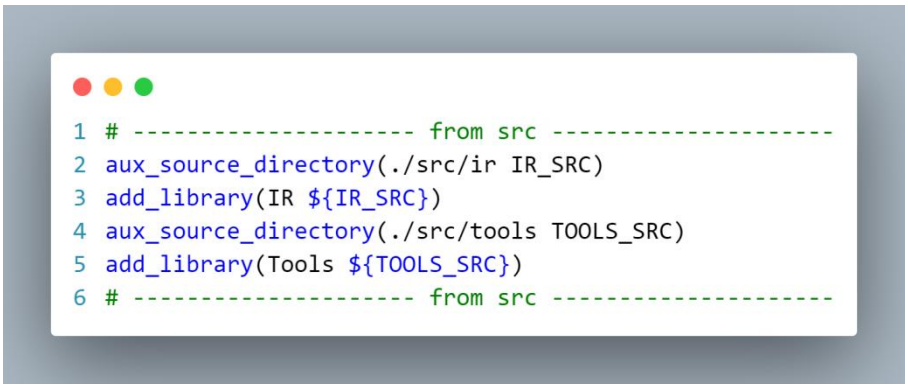


```
# ---------------------- from lib ----------------------
# link libxx.a
# u should rename libxx-x86-win.a or libxx-x86-linux.a
# to libxx.a according to ur own platform
link_directories(./lib)
# ---------------------- from lib ----------------------
```

在这种情况下，**Tools** 和 **IR** 被假定为预编译好的静态库（**libTools.a** 和 **libIR.a**），存放在 **lib** 目录下。通过 **link_directories(./lib)**命令指定静态库目录，然后通过 **target_link_libraries** 命令链接这些库。

2.2 使用源代码来链接库

通过将源代码文件编译成库，然后链接时使用这些库，这种方法在开发阶段非常有益，可以方便的进行调试，在 **CmakeLists.txt** 中通过如下代码直接链接静态库，如图：



```
# ---------------------- 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 ----------------------
```

在这种情况下，**Tools** 和 **IR** 库是从源代码文件编译而来的。通过 **aux_source_directory** 命令将源代码目录中的所有源文件添加到库中，然后使用 **add_library** 命令创建这些库，最后通过 **target_link_libraries** 命令链接这些库。

3. 如何支持短路运算？

在代码执行过程中，短路运算是逻辑运算的一种特性，即：如果第一个操作数已经能够确定整个表达式的值，第二个以及后面的操作数就不会被计算。这对于提高性能和避免不必要的计算非常有用。

在支持短路运算时，实验中主要涉及下面两种情形：

- 逻辑与（**&&**）运算：如果第一个操作数为假（**false**），整个表达式结果为假，不需要计算后面的操作数。
- 逻辑或（**||**）运算：如果第一个操作数为真（**true**），整个表达式结果为真，不需要计算后面的操作数。

以逻辑与（**&&**）短路运算为例：

```
// 如果第一个操作数为0（取反为opposite = 1），跳过后面的操作数
// LAndExp -> EqExp [ '&&' LAndExp ]
// second_and_instructions表示第二个表达式LAndExp的指令
// 跳转到第二个表达式的指令的下一条指令
instructions.push_back(new Instruction({opposite, ir::Type::Int},
                       {},
                       {std::to_string(second_and_instructions.size() + 1), ir::Type::IntLiteral},
                       Operator::_goto));
```

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 **RSIC-V** 架构下，函数调用需要通过一系列步骤来实现，如：保存现场、恢复现场、参数传递、调用函数以及处理返回值，下面是具体的步骤和解释：

4.1 函数调用者 caller（函数调用过程）

- **保存现场：**在调用函数之前，**caller** 需要保存自己使用的寄存器，但是在实验中采用使用后立即存储方法，故任何一条指令执行完临时寄存器都不会被暂用，所以仅需要保存返回寄存器 **ra**，将这些寄存器压入栈中，以便函数返回时能够恢复现场。
- **参数传递：**将函数参数一次存放在对应的参数寄存器中，即 **a0 - a7**。如果函数个数参数个数超过 8 个，需要将剩余的参数压入栈中。
- **调用函数：**添加 **jal** 指令跳转到目标函数地址，**jal** 指令会将当前 **pc** 保存到 **ra** 寄存器中作为返回地址。
- **等待函数执行结束返回**
- **恢复现场：**从栈中恢复之前保存的寄存器值，实验中即 **ra**。

4.2 被调用函数（被调用函数执行过程）

- **分配栈空间：**为当前函数分配栈帧空间，用于存放函数执行产生的局部变量，同时更新栈指针 **sp**，使其指向新的栈顶。
- **获取参数：**从参数寄存器 **a0 - a7** 中获取函数参数，多余的部分需要从 **caller** 的栈中读取，接着保存在自己的栈帧空间中。
- **执行函数体**
- **返回值传递：**将函数的返回值存放在 **a0** 寄存器中。

- **回收栈空间：**恢复栈指针 **sp**，释放当前函数的栈帧空间。
- **返回调用者：**使用 **jr** 指令跳转回 **caller** 的位置,即 **ra** 寄存器保存的地址。

四、实验测试

1、测试程序是如何运行的？

CmakeLists.txt 配置了整个项目的结构和编译规则，使用 **cmake** 和 **make** 命令进行项目构建进而得到可执行文件 **comiler**，运行 **compiler** 程序，传入对应的测试用例文件作为输入，得到输出结果，并于参考结果进行比对。

2、执行了什么命令？

- 启动 Docker 容器

  docker run -it -v (实验包路径):/coursegrader frankd35/demo:v3

- 进入 test 目录

  cd /coursegrader/test

- 编译构建项目

  python3 build.py

- 执行程序生成输出结果

  python3 run.py [s0 s1 s2 S]

- 比对参考结果

  python3 score.py [s0 s1 s2 S]

- 其他命令

  python3 test.py [s0 s1 s2 S] 执行编译、运行、比对步骤

  ../bin/compiler ./testcase/[basic/function]/xxx.sy -e -o test.out 执行单个样例

3、你的汇编是如何变成 RISV 程序并被执行的?

根据 **RISC-V** 中文手册中的 **ABI**，将实验二生成的中间标识 **IR** 逐条转换为 **RISC-V** 汇编，并且加入了数据定义、函数调用时保留现场和函数返回时还原现场等内容，生成 **RISC-V** 汇编后使用命令：

riscv32-unknown-linux-gnu-gcc ./test.s sylib-riscv-linux.a -o ./test.exe

riscv32-unknown-linux-gnu-gcc 指定编译器编译汇编程序生成可执行文件，接着使用命令：

qemu-riscv32.sh ./test.exe > ./test.out

qemu 可以模拟 **CPU** 和内存，运行 **qemu-riscv32.sh** 脚本在 **qemu** 上运行 **risc-v** 程序，输出结果。最后对照结果并打分使用的是 **diff**。

五、实验总结

1、实验过程中所遇到的问题及解决办法

1.1 实验一

① **问题：**没有处理注释代码，导致输出结果出错。

解决：提前去除注释代码，保证输入 **DFA** 的字符是有效的。

• • • • • •

1.2 实验二

① **问题：**c 代码中穿插着变量类型隐式转换，未处理导致结果错误。

解决：处理变量类型隐式转换。（未解决）

② **问题：未**进行短路运算导致输出结果不一致。

解决：添加短路运算。

• • • • • •

1.3 实验三（近期刚完成实验三，所以问题记得比较清）

① **问题：RISC-V** 中的 **and** 和 **or** 运算是按位与操作的，在判断语句中 如果直接将运算结果与 **0** 或者 **1** 比较的话会出现错误。

解决：使用 **snez** 指令将两个操作数置 **0** 或置 **1**，在进行 **and** 或者 **or** 运算。

② **问题：**实验中的 **RISC-V** 中的立即数范围是**-2048 - 2047**，即部分测试样例会出现溢出现象。

解决：对于所有的立即数使用 **li** 指令加载至寄存器中，在进行其他运算。

③ **问题：.word** 0 只能分配一个 4 字节空间，**.size** 关键词不进行空间的分配，导致数组变量仅被分配了 **4** 字节空间。

解决：对于数组变量使用 **.word 0, 0, 0・・・**。

④ **问题：**数组变量使用普通传值处理出错。

解决：数组变量应该传递其地址，同时访问时应该进行解引用。

・・・・・・

2、对实验的建议

原本打算利用近一个月的课余时间来解决浮点数相关的问题,但由于期末复习、夏令营准备以及学校的其他考核任务,导致实验三最后几天才得以完成。个人感觉课本知识并不太难掌握,关键在于实验部分,尤其是实验二和实验三,调试过程较为困难,有时甚至需要通过修改测试样例来寻找错误。我建议可以像实验一那样,将每个实验都采用作业和实验相结合的方式进行。同时,可以在 **GitHub** 上开设一个讨论区（像大数据课程那样）,收集每一个测试点的实现关键或者遇到的 **bug**,便于我们更好地定位程序错误。此外,也需要完善实验指导书的内容,详细地解释示例,以帮助我们更好地在完成实验任务的同时注意到一些需要考虑的问题。