**电子科技大学 信息与软件工程 学院**

**标 准 实 验 报 告**

**（实验）课程名称 编译技术**

**电子科技大学教务处制表**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：成棋伟 学 号：2022130102013**

**指导教师：周尔强**

**实验地点：信软楼 实验时间：2024.12.7**

**一、实验室名称：信软 303**

**二、实验项目名称：代码生成**

**三、实验学时：8学时**

**四、实验目的、原理、内容及步骤：**

**（一）实验任务：**

（1）查看指令手册LLVMRef.mht

（2）用clang学习编写简单的 C语言程序 test.c

用 clang -emit-llvm -S ./test.c 生成该文件对应的llvm指令

（3）学习掌握LLVM优化阶段的基本概念，不仅理解如何生成LLVM IR，还 要了解如何进行IR优化。

（4）掌握通过AST（抽象语法树）节点生成LLVM代码，理解如何从抽象语 法树（AST）中提取不同类型的语法节点，熟悉节点的分类和结构。

（5）在实验3及示例程序的基础上完成算术表达式、逻辑表达式、赋值语 句、条件语句、循环语句；

对应的代码生成，函数命名分别为：

genArithmeticExpr, genLogicExpr

genAssignStmt, genIfStmt, genWhileStmt

函数参数都为： (past node, char\* result)

node为相关类型的结点，

result 为用来保存三地址代码/LLVM虚拟指令的存储空间每行只放一条指令

**（二）实验原理**

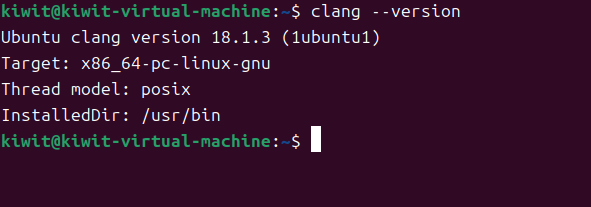
LLVM是一个开源的编译工具，提供了一种跨平台的中间代码（LLVM IR），可以通过Clang编译器把C语言代码转成这种中间代码。在这个实验中，我们会写一个代码生成器，把程序的语法树（AST）转换成相应的LLVM指令，模拟编译器的代码生成过程，帮助我们更好地理解编译的工作原理。

**（三）任务学习**

通过对提供资料以及网上相关文档的学习，我简单了解到LLVM的基本指令，同时也大致了解了clang和LLVM的关系——它是一个基于 LLVM 架构的 C 语言家族（C / C ++ / Objective-C）编译器前端。

安装clang（前面默认安装好了flex,Bison)：

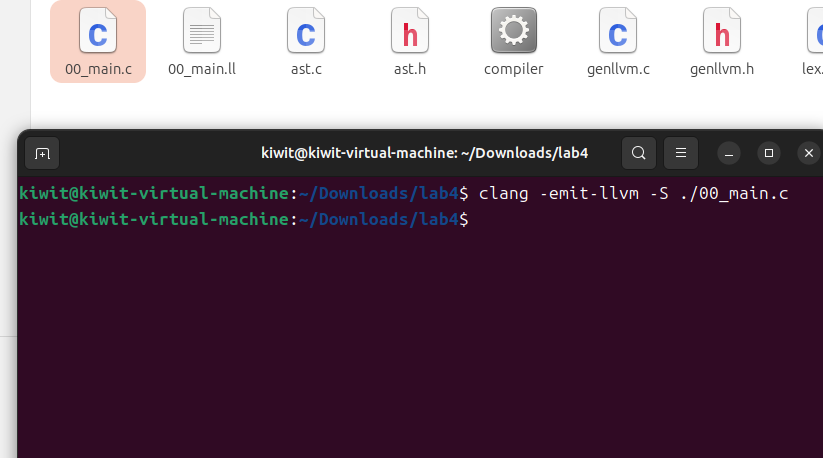
安装后输入 *clang--version,*查询是否在本地虚拟机成功安装环境



之后，在C程序00\_main.c对应文件目录的终端下输入命令：

*clang -emit-llvm -S ./00\_main.c*

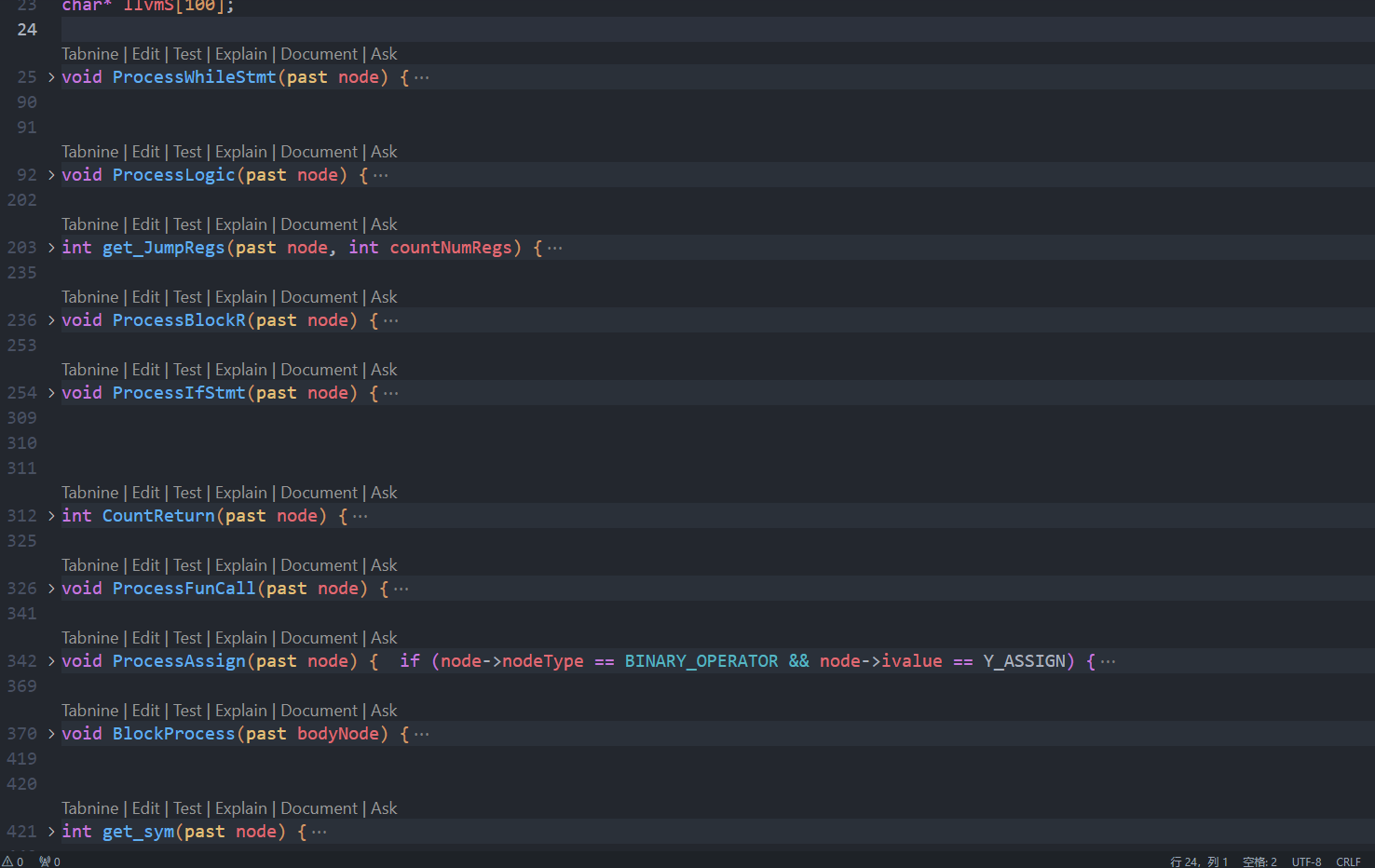
这样，clang 会读取 test.c 文件，对其进行语义分析和优化，然后输出 LLVM IR 的汇编表示形式，最终得到的中间表示代码（IR）名称为00\_main.ll。



可以看到，文件中间的部分就是自动生成的中间代码，



1. **函数代码编写**



**①算术表达式生成：**

在genArithmeticExpr函数中分类讨论，逐步解析表达式中的运算符（如+,-,\*,/）；之后递归地解析运算符两侧的子表达式。根据运算符生成相应的中间代码（三地址代码）。此外，还要注意管理操作数栈，以保持操作数的顺序和计算的正确性。

**②逻辑表达式生成：**

这里主要处理逻辑上的与（&&）、或（||）等逻辑表达式，在genLogicExpr函数中，我们考虑确定整个表达式的结果之后不再计算无用的表达式部分，也就是“短路”逻辑；然后我们递归地解析逻辑运算符两侧的子表达式；最后生成比较指令（如 icmp）和根据比较结果生成条件跳转指令（如 br）。

**③生成赋值语句的中间代码：**

genAssignStmt函数中，首先解析等号右侧的表达式，生成其计算结果。如果右侧结果是变量，生成加载指令；如果是常量，可以直接使用；最后将计算结果存储到左侧变量的位置，生成存储（store）指令。

**④生成条件语句的中间代码：**

这部分我们在genIfStmt函数实现。首先解析if语句的条件表达式，并生成其计算结果。根据条件结果生成条件跳转指令，如果有对应语句的话就跳转。比如：then部分或else部分。再之后我们递归地生成if语句then部分的代码，如果有else部分，也递归地生成该部分的代码。最后在末尾生成跳转指令，回到程序的主流程。

**⑤生成循环语句的中间代码：**

逻辑是：解析while循环的条件，并生成其计算结果。在条件为真时，进入循环体，递归地生成循环体内的代码；循环体执行完毕后，转回条件判断，继续循环或退出循环；再生成跳转：使用跳转指令（如br）来实现循环的控制流。

**（四）完整的代码生成逻辑**

**语法分析和AST构建：**

实验的第一步涉及使用词法分析器和语法分析器来解析源代码。词法分析器将源代码分解为一系列基本的词法单元，例如关键字、标识符、数值和运算符。法分析器根据这些词法单元和预定义的语法规则构建出AST。AST是源代码逻辑和结构的树形表示，其中每个节点代表了源代码中的一个构造，如表达式、语句或函数定义。

**AST遍历与节点处理：**

在AST构建完成后，实验接着要求遍历这棵树。遍历过程中，根据节点类型（如变量声明、if语句、while循环、函数声明等）来执行特定的操作。每种类 型的节点可能需要不同的处理方式，包括计算表达式的值等。

**代码生成和LLVM虚拟指令翻译：**

实验的核心部分是将AST中的每个节点转换为LLVM虚拟指令。对于不同类型的节点，需要开发相应的翻译函数。例如，一个if语句节点的翻译函数将生成条件分支的LLVM指令，而while循环节点的翻译函数则生成循环控制的指令。

**五、实验运行结果：**

**（一）测试环境：**

Ubuntu clang version 18.1.3 (1ubuntu1)

Target: x86\_64-pc-linux-gnu

Thread model: posix

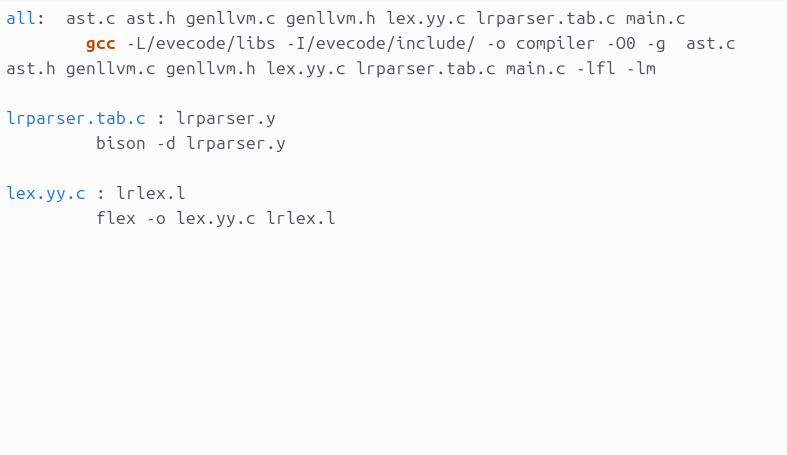
InstalledDir: /usr/bin

代码测试平台：icoding

1. **测试过程：**

**编写makefile文件：**

我们在前面的实验一、二、三中安装了flex、bison,学习LLVM指令时又安装了clang,为了使整个项目能完整地编译，产生一个能够生成中间代码IR的程序，我们需要编写一个makefile文件。通过定义规则实现对项目中所有源代码文件的自动化编译与链接，指定使用 gcc 编译器，并调用 flex 和 bison 工具生成词法分析器和语法分析器代码，从而定义目标文件、源文件以及各步骤的依赖关系。

Makefile文件编写如下：

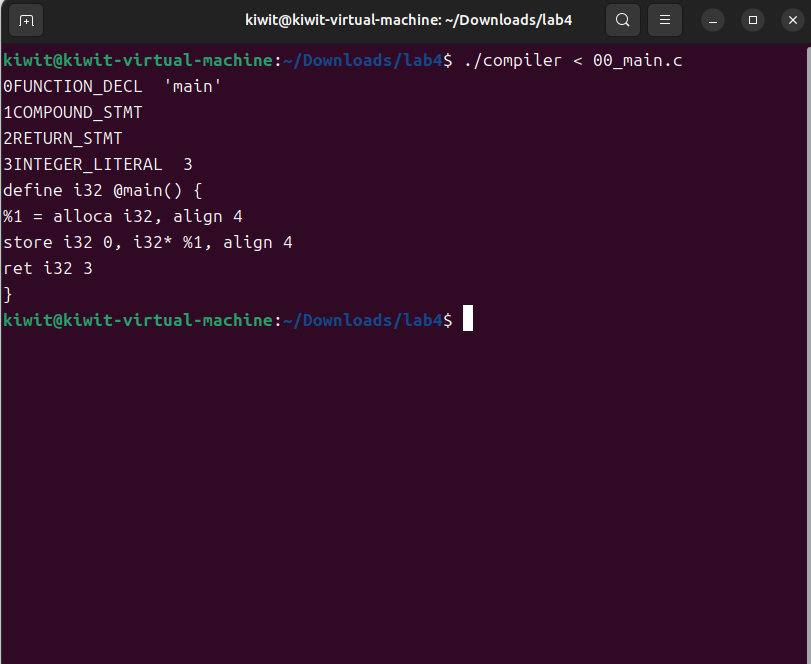
**文件夹目录下的命令行中输入**：

*Make*

编译产生了名为“compiler”的编译器，之后再在该目录下输入：

*./compiler <00\_main.c*

调用我们生成的编译器，即可得到对应测试用例00\_main.c的中间代码，如下图所示：



对比我们一开始得到的**00\_main.ll**文件，中间代码生成部分内容一致，说明编译器compiler能够正确运行产生中间代码，达到了预期的实验结果。

1. **对文件的解释：**

* ast.c和ast.h：

定义AST的数据结构和操作。AST（抽象语法树）是代码编译的中间表示，通过节点表示代码的各个结构。ast.h和 ast.c中提供了节点类型的定义和相关的创建函数如 new\_expr、new\_id、new\_func\_def 等，通过 show\_ast()

* genllvm.c和genllvm.h：

主要部分的代码，包含生成LLVM IR的逻辑。本实验中，我们使用 genllvm.c 来根据 AST 生成对应的 LLVM IR。关键函数如genllvm()：遍历 AST，依次调用相关的生成函数来生成 LLVM IR 代码；GCD\_CONST\_DECL()：处理常量声明，生成 constant 类型的 LLVM IR 代码。

* lrlex.l：
* Flex的词法分析器定义文件。
* lrparser.y：
* Bison的语法分析器定义文件。
* main.c：
* 程序的入口点，调用词法和语法分析器。
* lrparser.tab.h 和 lrparser.tab.c：
* 由bison生成，包含词法分析器和语法分析器的代码，它们从源代码中解析出AST。

lex.yy.c:

由Flex 工具生成，对应词法分析。

**六、实验结论与总结：**

整四个实验来看，编译过程从词法分析开始，通过将源代码文本转换为词法单元，接着使用递归下降或LR语法分析来解析这些单元并构建抽象语法树（AST）。递归下降解析适合于简单的语法，通过为每个语法规则编写函数进行处理，而LR分析能更有效地处理复杂语法，使用状态机和分析表来管理符号和规则。最后，在代码生成阶段，AST或中间表示被转换为目标代码，如LLVM中间表示。

**报告评分：**

**指导教师签字：**