华中科技大学

编译原理实验报告

专　　业：　计算机科学与技术

班　　级：　CS2207

学　　号：　U202215561

姓　　名：　瞿明睿

电　　话：　13956929603

邮　　箱：　3021018823@qq.com

独创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签名： 

日期：2025 年 6 月 22 日

综合成绩	
教师签名	

目 录

1 编译工具链的使用

1.1 实验任务

掌握 GCC/Clang 工具链的核心工作流程，包括：预处理、编译、汇编、链接四个阶段；高级编译选项（条件编译、优化级别、目标架构指定）；交叉编译配置；Makefile 自动化构建。

1.2 实验实现

用 gcc 连编 def-test.c, alibaba.c，生成 def-test 二进制可执行代码

通过以前的经验可以得到：gcc def-test.c alibaba.c -o def-test

同时注意到 def-test.c 中#ifdef BILIBILI 字段

```
#include <stdio.h>
#include "alibaba.h"
int main(void)
{
    printf( "Instructor: Hello, I am your instuctor, please introduce yourself.\n");

    #ifdef BILIBILI
    printf("BILIBILI: My name is Bili, 先生お久しぶりです!\n");
    #endif

    alibaba();
    return 0;
}
```

则修正为 gcc -DBILIBILI def-test.c alibaba.c -o def-test

用 clang 编译 bar.c，需要 O2 优化， armv7 架构，linux 环境下，符合 gnueabihf 嵌入式二进制接口规则，并支持 arm 硬浮点的汇编代码，得到汇编代码文件名为 bar.clang.arm.s

查询即可得到：clang -S -O2 -target armv7-linux-gnueabihf -o bar.clang.arm.s bar.c

在交叉编译部分，根据要求对应的编译执行即可

使用 arm-linux-gnueabihf-gcc 将 iplusf.c 编译为 ARM 汇编代码

iplusf.arm.s

```
arm-linux-gnueabihf-gcc -S -march=armv7-a -mfpu=vfpv4 -mfloat-abi=hard -o iplusf.arm.s iplusf.c

if [ $? -ne 0 ]; then
    echo "Error: Compilation to assembly failed."
    exit 1
fi
```

使用 arm-linux-gnueabihf-gcc 将汇编代码 iplusf.arm.s 汇编并链接 SysY2022 运行时库 sylib.a，生成 ARM 可执行文件 iplusf.arm

```
arm-linux-gnueabihf-gcc -march=armv7-a -mfpu=vfpv4 -mfloat-abi=hard -o iplusf.arm iplusf.arm.s sylib.a

if [ $? -ne 0 ]; then
    exit 1
fi
```

使用 qemu-arm 运行生成的 ARM 可执行文件 iplusf.arm

```
qemu-arm -L /usr/arm-linux-gnueabihf ./iplusf.arm

if [ $? -ne 0 ]; then
    exit 1
fi
```

使用 make helloworld 来构建目标程序 helloworld，并运行 helloworld

```
helloworld: helloworld.o main.o
    g++ -o helloworld helloworld.o main.o

helloworld.o: helloworld.cc include/helloworld.hh
    g++ -c helloworld.cc -Iinclude

main.o: main.cc include/helloworld.hh
    g++ -c main.cc -Iinclude

clean:
    rm -f helloworld *.os
```

这里使用偷懒的方法直接针对所有的文件直接使用 g++输出，可以这样改写，使用模块化变量化的设计，更加规范

```
CC = g++
CFLAGS = -Iinclude  # 指定头文件目录
TARGET = helloworld
OBJS = helloworld.o main.o

all: $(TARGET)
```

```
$(TARGET): $(OBJS)
	$(CC) -o $@ $^

helloworld.o: helloworld.cc include/helloworld.hh
	$(CC) -c $(CFLAGS) $< -o $@

main.o: main.cc include/helloworld.hh
	$(CC) -c $(CFLAGS) $< -o $@

clean:
	rm -f $(TARGET) $(OBJS)

.PHONY: clean all
```

其中需要注意 makefile 容易出现空格的问题，需要注意 tab 和空格的使用。

2 词法分析

2.1 实验任务

词法分析阶段作为编译器的第一个阶段，核心是从源程序中识别出具有独立意义的 token。关键任务：匹配关键字、标识符、常量、运算符和界符等 token；处理空白符和注释在识别过程中将其忽略；错误检测与报告。

2.2 词法分析器的设计与实现

实验利用 Flex 工具构建词法分析器。Flex 是一个用于生成词法分析器的工具，它读取一个包含正则表达式和相应动作的输入文件，然后生成一个 C/C++ 源文件，该文件包含了词法分析器的实现。

Flex 源文件通常分为三个部分，以下给出，由`%%`分隔：

```
%{
//辅助定义部分
// C/C++ 代码，会被直接复制到生成文件的开头，例如：头文件包含、全局变量声明等
%}
name definition   // 正则表达式定义（宏定义）
%%
// 规则部分，匹配模式（正则表达式）及其对应的动作（C/C++ 代码）
pattern1 { action1 }
pattern2 { action2 }
%%
// 用户子程序段，辅助函数等 C/C++ 代码，会被复制到生成文件的末尾
```

实验在这里的设计如下

辅助定义部分定义了一些常用的正则表达式：

```
ID [a-z_A-Z][a-z_A-Z0-9]*
INT ([1-9][0-9]*|0[0-7]*|(0x|0X)[0-9a-fA-F]+)
EXP ([Ee][-+]?[0-9]+)
FLOAT (([0-9]*\.[0-9]+|[0-9]+\.){EXP}?[fF]?)|[0-9]+{EXP}[fF]?
FLOAT_LIT ({DEC_FLOAT_LIT}|{HEX_FLOAT_LIT})
```

规则部分定义了如何匹配这些正则表达式以及匹配后的动作：

```
"int"{return INTTYPE;}
"float"{return FLOATTYPE;}
"void"{return VOID;}
{ID} {yylval.token = new string(yytext); return ID;}
{INT}{yylval.int_val = strtol(yytext,nullptr,0); return INT;}
```

```
{FLOAT_LIT}{yylval.float_val = strtof(yytext,nullptr); return FLOAT;}
```

对于空白符和注释，定义了匹配后执行空动作{}，表示忽略这些内容：

```
[\n] {yycolumn=1;}
[ \r\t] {}
{SingleLineComment}{}
{MultilineComment}{}
```

对于无法匹配任何规则的字符，会报告词法错误：

```
. {printf("Error type A :Mysterious character \"%s\"\n\t at Line %d\n",yytext,yylineno);}
```

具体实验中因为不了解 flex 的语法，导致了很多次的编译无法通过，后面通过查询文档可以顺利解决，同时需要注意，不同的正则表达式可能匹配同一段输入，导致错误无法通过

3 语法分析

3.1 实验任务

语法分析阶段是编译器的第二个阶段，其主要任务是根据语言文法规则，对词法分析产生的 token 序列进行解析，验证输入程序是否符合语法结构。同时，在语法正确的情况下，构建程序的 AST，为后续的语义分析和代码生成阶段提供结构化的表示。

3.2 语法分析器的设计与实现

实验利用 Bison 工具构建语法分析器。Bison 是一个通用的解析器生成器，它读取一个包含上下文无关文法规则的输入文件，然后生成一个 C/C++ 源文件，这一点和 flex 是一样的。所以通常结合使用，Flex 负责词法分析而 Bison 负责语法分析。同时也有相似的源文件结构：

Bison 源文件通常也分为三个部分，由%%分隔：

```
%{//辅助定义部分
  // C/C++ 代码，会被直接复制到生成文件的开头例如：头文件包含、token 声明、语义值类型定义等
%}
// Bison 声明，如 token 声明、非终结符类型声明、优先级和结合性声明等
%token TOKEN_NAME
%type <type> non_terminal
%%
//文法规则部分，上下文无关文法产生式及其对应的语义动作
non_terminal: production { semantic_action }
            | another_production { another_action }
            ;
%%
//用户子程序段，辅助函数等 C/C++ 代码，会被复制到生成文件的末尾
```

具体到实验设计，关注的是文法规则方面，根据 SysY2022 语言的文法定义，将产生式转换为 Bison 规则。每个规则后面可以跟随一个用{}括起来的 C++代码块语义动作。这些动作在规则被归约时执行用于构建 AST 节点。

```
stmt: lval ASSIGN exp SEMICOLON {
        $$ = new StmtAST(STYPE::ASS, $1, $3);
    }
```

```
    | exp_opt SEMICOLON {
        $$ = new StmtAST(STYPE::EXP, $1);
      }
    | block {
        $$ = new StmtAST(STYPE::BLK, $1);
      }
    | IF LP cond RP stmt {
        $$ = new StmtAST(STYPE::SEL, $3, $5);
      }
    | IF LP cond RP stmt ELSE stmt {
        $$ = new StmtAST(STYPE::SEL, $3, $5, $7);
      }
    | WHILE LP cond RP stmt {
        $$ = new StmtAST(STYPE::ITR, $3, $5);
      }
    | BREAK SEMICOLON {
        $$ = new StmtAST(STYPE::BRE);
      }
    | CONTINUE SEMICOLON {
        $$ = new StmtAST(STYPE::CONT);
      }
    | RETURN exp_opt SEMICOLON {
        $$ = new StmtAST(STYPE::RET, $2);
      }
    ;
```

在语义动作中，$$表示当前产生式左部非终结符的语义值，$1， $2， $3 等表示产生式右部各个符号的语义值。通过 new 操作创建 AST 节点，并将子节点的指针作为参数传递，从而构建 AST 的层次结构。

4 静态语义分析

4.1 实验任务

完成 checker.cpp 设计，检查包括：检查所有使用的变量和函数是否已经声明，以及是否存在重复声明的情况。不同作用域的嵌套和查找规则；检查各种操作中操作数的类型是否兼容，并在需要时处理隐式类型转换；控制流语句的使用是否合法，如 break 和 continue 是否出现在循环体内；其他语义规则检查和错误报告。

4.2 静态语义分析的设计与实现

本实验采用访问者模式来实现静态语义检查器。访问者模式允许在不修改 AST 节点类结构的情况下，对 AST 中的每个节点执行特定的操作。我们定义一个语义检查器类 Checker，它继承自抽象访问者类 Visitor，并主要任务是完成不同的多个 visit 方法来处理不同类型的 AST 节点。

其核心是数据结构定义在 checher.h

```
struct Entry {
bool is_array{};                                    //是否为数组
bool is_func{};                                     //是否为函数
TYPE type;                                          //变量类型或返回值类型
int array_length{};                                 //数组维度
std::vector<int> arlen_value;                       //数组每一维长度
std::vector<struct Entry> func_params; //函数参数列表
};
```

同时.h 文件中定义了符号表管理函数，如 make_new_table()创建新作用域，具体到.cpp 实现的时候核心使用是需要如变量处理 InsertVar()，函数处理 InsertFunc()，以及非常重要的 AST 遍历 visit 方法族，不多做赘述

同时错误检测结构定义如下

```
enum class ErrorType {
VarUnknown = 1,                 //使用未定义变量
VarDuplicated,                  //变量重复定义
FuncUnknown,                    //使用未定义函数
FuncDuplicated,                 //函数重复定义
FuncParamsNotMatch,             //函数参数不匹配
```

```
FuncReturnTypeNotMatch,   //函数返回值类型不匹配
ArrayIndexNotInt,         //数组下标不是整数
BreakNotInLoop,           //break 不在循环中
ContinueNotInLoop,        //continue 不在循环中
VisitVariableError        //非数组变量使用下标访问
};
```

错误检测流程：在 AST 节点遍历过程中发现语义错误时，调用 err.error 报告错误，并终止编译过程。

具体实现的检查方法分为几个，简要介绍：

变量与定义：也就是 InsertVar 方法中实现，遍历变量定义列表，对每个变量创建 Entry 结构，记录类型、是否为数组等信息使用插入符号表，若插入失败报告错误

```
auto [it, inserted] = table.front().try_emplace(*def->id, tmp);
if (!inserted) {
err.error(ErrorType::VarDuplicated, *def->id);
return false;
}
```

函数检查：我们需要检查函数是否重复定义，同时比较实参与形参的数量和类型是否匹配，最后比较返回值类型与函数声明类型是否一致

```
// 参数数量检查
if (entry->func_params.size() != ast.funcCParamList.size()) {
err.error(ErrorType::FuncParamsNotMatch, *ast.id);
exit(int(ErrorType::FuncParamsNotMatch));
}
// 参数类型检查
for (auto &exp : ast.funcCParamList) {
exp->accept(*this);
if (this->current_type.type != entry->func_params[i].type) {
      err.error(ErrorType::FuncParamsNotMatch, *ast.id);
    exit(int(ErrorType::FuncParamsNotMatch));
 }
}
```

数组：数组下标必须为整数类型，非数组变量不能使用下标访问

```
// 数组下标是否为整数检查
for (auto &exp : ast.arrays) {
if (exp) {
   exp->accept(*this);
   if (!this->Expr_int) {
      err.error(ErrorType::ArrayIndexNotInt, *ast.id);
      exit(int(ErrorType::ArrayIndexNotInt));
   }
```

```
}
}
// 非数组变量使用下标访问检查
if (!entry->is_array && !ast.arrays.empty()) {
err.error(ErrorType::VisitVariableError, *ast.id);
exit(int(ErrorType::VisitVariableError));
}
```

控制流语句：使用标志跟踪当前是否在循环中，遇到 break 或 continue 时检查 in_loop 标志

```
if (ast.sType == STYPE::BRE) { // Break 语句检查
if (!this->in_loop) {
    err.error(ErrorType::BreakNotInLoop, current_func_name);
    exit(int(ErrorType::BreakNotInLoop));
  }
} else if (ast.sType == STYPE::CONT) { // Continue 语句检查
if (!this->in_loop) {
    err.error(ErrorType::ContinueNotInLoop, current_func_name);
    exit(int(ErrorType::ContinueNotInLoop));
  }
}
```

更多的实现过长不过多展示，但是静态语义检查非常重要，正确的类型信息确保中间代码生成正确的运算指令

5 中间代码生成

5.1 实验任务

中间代码生成阶段是编译器的重要环节，它将经过静态语义检查的 AST 转换为一种与源语言和目标机器都无关的中间表示。本实验选择 LLVM IR 作为中间代码。生成的 LLVM IR 必须精确地保留源程序的所有语义信息，确保程序的行为在转换前后保持一致。将 AST 的层次结构和控制流信息转换为 LLVM IR 的模块、函数、基本块和指令结构。

5.2 中间代码生成器的实现

实验通过遍历 AST 并利用 LLVM 提供的 API 来生成 LLVM IR。我们实现了一个 GenIR 类，它继承自 Visitor 类，并重写了各种 visit 方法，用于处理不同类型的 AST 节点并生成相应的 LLVM IR 指令。

具体到实验添加的方法，通过一个 switch 语句根据语句类型 ast.sType 分别处理不同类型的语句，例如分号语句、表达式语句、赋值语句、continue 语句、break 语句、return 语句、块语句、if 语句和循环语句。

其中，赋值语句 ASS 的处理是中间代码生成中的一个关键部分。根据实验要求，处理赋值语句需要完成以下步骤：

获取左值的地址：赋值语句的左值代表一个存储位置，在访问左值对应的 AST 节点之前，需要设置一个来指示 visit 方法返回变量的地址（指针）而不是其存储的值。访问标志 ast.lVal 后，获取返回的 Value 对象，即左值的地址。

获取右值的值：赋值语句的右值代表一个需要计算出结果的表达式。访问右值对应的 AST 节点，获取返回的 Value 对象，即右值表达式的计算结果。

类型检查和转换：LLVM IR 是强类型语言，赋值操作要求左值（指针指向的类型）和右值的值类型一致。因此，需要检查左值指针指向的类型与右值的值类型是否匹配。如果类型不匹配，需要根据语言的规则进行隐式类型转换。例如，将浮点数转换为整数使用，将整数转换为浮点数使用。

生成存储指令：使用 IRStmtBuilder::create_store 方法生成 store 指令，将右值的值存储到左值所指向的内存地址。

最后实验添加代码按照赋值的要求写即可

```
case ASS: {
    // 确保左值和表达式存在
    if (!ast.lVal || !ast.exp) break;
    // 获取左值地址
    requireLVal = true;
    ast.lVal->accept(*this);
    Value* lval_ptr = recentVal;  // 保存左值地址
    requireLVal = false;  // 重置标记
    // 获取右值表达式
    ast.exp->accept(*this);
    Value* rval = recentVal;  // 保存右值
    // 获取左值指针指向的实际类型
    Type* lval_type = static_cast<PointerType*>(lval_ptr->type_)->contained_;
    // 类型不匹配情况
    if (lval_type != rval->type_) {
        if (lval_type == INT32_T && rval->type_ == FLOAT_T) {
            rval = builder->create_fptosi(rval, INT32_T);
        } else if (lval_type == FLOAT_T && rval->type_ == INT32_T) {
            rval = builder->create_sitofp(rval, FLOAT_T);
        }
    }
    // 存储指令
    builder->create_store(rval, lval_ptr);
    break;
}
```

这就是赋值语句在生成 LLVM IR 时的处理流程，确保生成代码符合 LLVM 的类型系统要求，并将表达式计算结果正确地存储到变量对应的内存位置。其他语句类型（如控制流语句、函数调用等）的 LLVM IR 生成则由 visit 方法中的其他 case 分支或调用的其他 visit 方法完成。

6 目标代码生成

6.1 实验任务

目标代码生成阶段是编译器的最后一个主要阶段，其任务是将中间代码翻译成特定目标机器体系结构上的机器代码或汇编代码。本实验的目标是利用 LLVM 工具链，将之前生成的 LLVM IR 中间代码翻译成 RISC-V 架构的汇编代码。生成正确的目标代码：将 LLVM IR 指令准确地映射到 RISC-V 指令集，确保生成的汇编代码能够正确执行源程序的逻辑。

6.2 目标代码生成器的设计与实现

初始化目标信息，在使用 LLVM 后端之前，需要初始化所有可用的目标信息。这通过调用一系列的初始化函数来完成：

```
// 初始化所有目标信息
InitializeAllTargetInfos();
InitializeAllTargets();
InitializeAllTargetMCs();
InitializeAllAsmParsers();
InitializeAllAsmPrinters();
```

目标三元组是一个字符串，用于描述目标机器的体系结构等参数。由于输入的 LLVM IR 文件可能不包含目标信息，我们需要在 Module 对象中设置目标三元组。

```
// 设置目标三元组为 RISC-V 64 位
auto target_triple = "riscv64-unknown-elf";
module->setTargetTriple(target_triple);
```

查找目标并创建目标机器：根据设置的目标三元组，使用函数查找对应的目标。后使用目标对象创建 TargetMachine（提供目标平台的完整机器描述等）

```
// 查找目标
std::string Error;
auto Target = TargetRegistry::lookupTarget(target_triple, Error);
if (!Target) {
    llvm::errs() << Error;
    return false;}
 // 创建目标机器
auto CPU = ""; //需要设置为默认
```

```
auto Features = ""; // 不使用特定的 CPU 特性
TargetOptions opt;
auto RM = Optional<Reloc::Model>();
auto TheTargetMachine =
Target->createTargetMachine(target_triple, CPU, Features, opt, RM,
Optional<CodeGenOpt::Level>());
module->setDataLayout(TheTargetMachine->createDataLayout()); // 设置数据布局
```

同时因为作用域的问题，使用 PassManager 来组织和运行 LLVM pass。调用 TargetMachine 的 addPassesToEmitFile 方法，将生成目标代码的 pass 添加到 PassManager 中。

```
auto gen_filename = getGenFilename(ir_filename, gen_filetype); // 获取输出文件名
std::error_code EC;
raw_fd_ostream dest(gen_filename, EC, sys::fs::OF_None);
if (EC) {
    llvm::errs() << "Could not open file: " << EC.message();
    return false;
}
legacy::PassManager pass; // 创建 PassManager
if (TheTargetMachine->addPassesToEmitFile(pass, dest, nullptr, gen_filetype)) {
    llvm::errs() << "TheTargetMachine can't emit a file of this type\n";// 添加生成目标代码的
pass
    return false;}
// 运行 PassManager 生成目标代码
     pass.run(*module);
     dest.flush(); // 确保所有内容写入文件
```

通过以上步骤，codeGenerate 函数能够将输入的 LLVM IR 成功翻译成 RISCV 架构下的目标代码。

7 总结

7.1 实验感想

通过本次实验，我对编译器的各个阶段有了更深入的理解。从词法分析到目标代码生成，每个阶段都有其独特的技术挑战。在实验过程中，我遇到了不少困难，例如在语法分析阶段处理复杂的文法规则时，需要仔细设计 AST 结构；在中间代码生成阶段，类型转换和 LLVM IR 指令的选择也花费了不少时间。但是，通过查阅资料和调试，我逐步解决了这些问题。实验设计覆盖了编译器构建的核心步骤，但部分实验指导可以更详细一些，例如在目标代码生成阶段，更多的示例会更有帮助。

7.2 实验总结与展望

本次实验让我系统地实践了编译器的构建过程，掌握了 Flex、Bison 和 LLVM 等工具的使用。我学会了如何设计词法规则和文法规则，如何构建 AST 并进行语义分析，以及如何生成中间代码和目标代码。在实验过程中，我深刻理解了编译器各阶段之间的衔接关系。未来，我希望在优化方面进行更深入的学习，例如中间代码的优化和目标代码的优化。此外，我也计划支持更多的语言特性，如指针和结构体。