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：　CS2007

学　　号：　U202015489

姓　　名：　蔡济舟

电　　话：　13683356870

邮　　箱：　cjz210oliver@gmail.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：蔡济舟

日期：2023 年　6 月 5 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

(1) 编译工具链的使用；

(2) Sysy 语言及运行时库；

(3) 目标平台 arm 的汇编语言；

(4) 目标平台 riscv64 的汇编语言；

以上任务中(1)(2)为必做任务，(3)(4)中任选一个完成即可。

1.2 实验实现

（1）编译工具链的使用

①GCC 编译器的使用

gcc 是一组跨平台，支持多语言的编译器套件，在之前的汇编，系统结构课程中已经有所了解。本关需要用 gcc 编译两个 c 语言程序，生成可执行程序。这里简单介绍通过本关的必要命令选项：-o 后面的参数指出生成的可执行文件的名称；-D 选项则完成宏定义的功能，以实现 Alibaba 和 Bilibili 之间的“对话”。关键代码如下：

```
gcc def-test.c alibaba.c -DBILIBILI -o def-test
```

②CLANG 编译器的使用

本关需要把 c 语言文件用 clang 编译器编译为汇编程序，同样没有难度，只需要了解相关命令选项即可：-O2 表示的是优化级别；-target 表明了生成的汇编程序需要遵循的架构和规范等；-S 表示生成汇编程序；-o 与 gcc 一样指定

生成文件名称。关键代码如下：

```
clang -O2 -S -target armv7-linux-gnueabihf bar.c -o bar.clang.arm.s
```

③交叉编译器 arm-linux-gnueabihf-gcc 和 qemu-arm 虚拟机

这一关与前面也无本质区别，只需阅读实验文档，替换相关命令，写好选项和参数，部分示例代码文档里也有，略加修改即可。关键代码如下：

```
# 用 arm-linux-gnueabihf-gcc 将 iplusf.c 编译成 arm 汇编代码 iplusf.arm.s
arm-linux-gnueabihf-gcc -S -o iplusf.arm.s iplusf.c
# 再次用 arm-linux-gnueabihf-gcc 汇编 iplusf.arm.s，同时连接 SysY2022 的运行时库 sylib.a，
生成 arm 的可执行代码 iplusf.arm
arm-linux-gnueabihf-gcc -o iplusf.arm iplusf.arm.s sylib.a
# 用 qemu-arm 运行 iplusf.arm
qemu-arm -L /usr/arm-linux-gnueabihf/ iplusf.arm
```

④make 的使用

Make 是一种用来进行项目构建的工具，个人感觉就是把命令行中的步骤统一写到一个 makefile 文件中，但是可以有一些额外的语法精简命令行数。首先是定义各种变量减少输入量：

```
CXX := g++
CXXFLAGS := -std=c++11 -Wall -Wextra -pedantic
INCLUDES := -I./include
OBJECTS := main.o helloworld.o
TARGET = helloworld
```

之后根据生成可执行程序的逆顺序依次写出指令，这里有几点要注意，变量用$()引用，%.cc 和%.o 能进行后缀匹配，^{和}@分别表示所有依赖文件和目标文件：

```
all: $(TARGET)
$(TARGET): $(OBJECTS)
  $(CXX) $(CXXFLAGS) $(INCLUDES) $^ -o $@
%.o: %.cc
  $(CXX) $(CXXFLAGS) $(INCLUDES) -c $^ -o $@
```

（2）Sysy 语言及其运行时库

由于 SysY 语言与 c 语言很相似，因此这关只需要实现一个计算最大利润的算法即可。思路如下：只需一遍遍历，维护一个目前最低股价和目前最大利润变量 minPrice 和 maxProfit，遍历时同步更新这两个变量，最后就能得到过程中的

最大利润，关键代码如下所示：

```
int maxProfit(int prices[]){
    // ----------  开始
    int minPrice = prices[0];
    int maxProfit = 0;
    int i=1;
    while(i<N){
        if (maxProfit < prices[i] - minPrice) {
            maxProfit = prices[i] - minPrice;
        }
        if (minPrice > prices[i]) {
            minPrice = prices[i];
        }
        i = i + 1;
    }
    return maxProfit;
    // ----------  结束
}
```

（3）arm 汇编

本关设计思路和 riscv 完全一样，只是语法有所区别，详细思路见 riscv 汇编部分。

（4）riscv 汇编

本关用 riscv 汇编实现一个升序的冒泡排序算法。已知数组首地址保存在 a0 中，元素个数保存在 a1 中，最后需要 a0 返回 0。程序开头首先养成良好习惯，把将要用到的寄存器旧值保存到栈中，赋值 a2=0 作为程序结束时的 a1 的值。a1 和 a2 控制外层循环的结束。a3 是依次遍历数组的元素指针，a4 和 a7 控制内层循环的结束，a5 和 a6 用于冒泡排序必要时交换相邻元素，临时存储元素。代码如下：

```
        .text
        .align  1
        .globl  bubblesort
        .type   bubblesort, @function
bubblesort:
  addi sp, sp, -24
   sw       a2, 0(sp)
```

```
    sw      a4, 4(sp)
    sw      a5, 8(sp)
    sw      a6, 12(sp)
    sw      a7, 16(sp)
    sw      ra, 20(sp)
    li      a2, 0
    addiw   a1, a1, -1
LOOPSTART:
    ble     a1, a2, FUNCEND
    mv      a3, a0
    li      a4, 0
    sub     a7, a1, a2
LOOPI:
    blt     a4, a7, LOOPJ
    addi a2, a2, 1
    j       LOOPSTART
LOOPJ:
    lw      a5, 0(a3)
    lw      a6, 4(a3)
    addi a4, a4, 1
    ble     a5, a6, SKIP
    sw      a6, 0(a3)
    sw      a5, 4(a3)
    addi a3, a3, 4
    j       LOOPI
SKIP:
    addi a3, a3, 4
    j       LOOPI
FUNCEND:
    li      a0, 0
    lw      a2, 0(sp)
    lw      a4, 4(sp)
    lw      a5, 8(sp)
    lw      a6, 12(sp)
    lw      a7, 16(sp)
    lw      ra, 20(sp)
    addi    sp, sp, 24
    ret

    .size   bubblesort, .-bubblesort
```

2 词法分析

2.1 实验任务

分别在给出的语法分析器框架的基础上，实现一个 Sysy 语言的语法分析器：

(1) 基于 flex 的 Sysy 词法分析器(C 语言实现)

(2) 基于 flex 的 Sysy 词法分析器(C++实现)

(3) 基于 antlr4 的 Sysy 词法分析器(C++实现)

以上任务任选一个完成即可。

2.2 词法分析器的实现

（1）基于 flex 的 Sysy 词法分析器(C 语言实现)

首先的工作是写出正确的正则表达式以匹配标识符，int 和 float 型字面量。标识符较为简单，是以字母或下划线开头，后面为字母数字或下划线的名称。对于 int 型，首先分别写出对于 8,10,16 进制数字字面量识别的正则表达式，最后用或语句连接，在数字的前面还有可能出现+-号或者没有，可以用[]表示任选其一，？表示出现或者不出现。*是 0 或任意次，+是 1 或任意次。Float 型字面量同理，总体可以分为三大类，用|表示的或语句连接不同格式即可。相关代码如下：

```
DIGIT [0-9]
LETTER [A-Za-z]
IDEN [A-Za-z_][A-Za-z0-9_]*

DEC_INTEGER ([0-9]+)
HEX_INTEGER 0[xX]({DIGIT}|[A-Fa-f])+
OCTAL_INTEGER 0[0-7]+
INTEGER [+-]?{DEC_INTEGER}|{HEX_INTEGER}|{OCTAL_INTEGER}

FLOAT [+-]?(([.][0-9]+([eE][+-]?[0-9]+)?[fF]?)|([0-9]+[.][0-9]*([eE][+-]?[0-9]+)?[fF]?)|([0-9]+[eE][+-]?[0-9]+[fF]?))
```

之后是对于错误报告的处理，这个由于 educoder 的示例均较为简单，我选

择面向输出编程，两处附加的错误处理如下所示：

```
{INTEGER} {
    if ('0' == yytext[0] && 'x' != yytext[1] && 'X' != yytext[1]) {
        for(char *c = yytext; *c != '\0'; c++) {
            if (*c < '0' || *c > '7') {
                printf("Lexical error - line %d : %s\n",yylineno,yytext);
                return LEX_ERR;
            }
        }
    }
    printf("%s : INT_LIT\n", yytext);
    return INT_LIT;
}

{INTEGER}{IDEN}  {printf("Lexical error - line %d : %s\n",yylineno,yytext);return LEX_ERR;}
```

（2）基于 flex 的词法分析器（C++ 实现）

不能说毫无区别，只能说一模一样，C 实现中添加的代码也添加到本关的相应位置即可。

（3）基于 antlr 的词法分析器（C++实现）

Antlr 只是语法上略有区别，比如对于词的识别要用冒号和分号表示起止，对于匹配单个字符要用单引号，其他几乎没有区别。在输出方面，只需要仔细阅读文档，了解 getText()，getType()等函数就可以轻松实现相关功能。下面只给出 float 字面量识别和处理输出的部分代码：

```
FLOAT_LIT      :      [+-]?(('.'[0-9]+([eE][+-]?[0-9]+)?[fF]?)|([0-9]+'.'[0-9]*([eE][+-]?[0-9]+)?[fF]?)|([0-9]+[eE][+-]?[0-9]+[fF]?));

for (auto token : tokens.getTokens()) {
        auto tokentype = token->getType();
        if(tokentype != lexer.EOF){
            if(tokentype == 43){
                std::cout<<"Lexical error - line "<<token->getLine()<<" : "<<
                    token->getText() << std::endl;
            }
            else  std::cout<<token->getText()  <<"  :  "  <<  tokenTypeName[tokentype]
<<std::endl;
        }
    }
```

3 语法分析

3.1 实验任务

分别在给出的语法分析器框架的基础上，实现一个 Sysy 语言的语法分析器：

(1) 基于 flex/bison 的语法分析器(C 语言实现)

(2) 基于 flex/bison 的语法分析器(C++实现)

(3) 基于 antlr4 的语法分析器(C++实现)

以上任务任选一个完成即可。

3.2 语法分析器的实现

（1）基于 flex/bison 的语法分析器(C 语言实现)

①语法检查

很简单嗷，代码注释给了语法，直接根据这个把相应符号换成对应的名称，甚至不需要构造抽象语法树就能过关，我哭死。

②语法分析

这一关就是在前面的基础上，用 new_node()函数构建抽象语法树结点，仿照其他部分的代码进行传参，需要额外注意传参的顺序和标注语句类型。代码如下：

```
Stmt: LVal ASSIGN Exp SEMICOLON {$$ = new_node(Stmt, NULL, $1, $3, AssignStmt, 0, NULL, NonType);}
    | Block {$$ = new_node(Stmt, NULL, NULL, $1, Block, 0, NULL, NonType);}
    | SEMICOLON {$$ = new_node(Stmt, NULL, NULL, NULL, BlankStmt, 0, NULL, NonType);}
    | Exp SEMICOLON {$$ = new_node(Stmt, NULL, NULL, $1, ExpStmt, 0, NULL, NonType);}
    | IF LP Cond RP Stmt ELSE Stmt {$$ = new_node(Stmt, $3, $5, $7, IfElseStmt, 0, NULL, NonType);}
    | IF LP Cond RP Stmt {$$ = new_node(Stmt, $3, NULL, $5, IfStmt, 0, NULL, NonType);}
    | WHILE LP Cond RP Stmt {$$ = new_node(Stmt, $3, NULL, $5, WhileStmt, 0, NULL, NonType);}
    | BREAK SEMICOLON {$$ = new_node(Stmt, NULL, NULL, NULL, BreakStmt, 0,
```

```
NULL, NonType);}
    | CONTINUE SEMICOLON {$$ = new_node(Stmt, NULL, NULL, NULL,
ContinueStmt, 0, NULL, NonType);}
    | RETURN Exp SEMICOLON {$$ = new_node(Stmt, NULL, NULL, $2, ReturnStmt,
0, NULL, NonType);}
    | RETURN SEMICOLON {$$ = new_node(Stmt, NULL, NULL, NULL,
BlankReturnStmt, 0, NULL, NonType);};
```

（2）基于 flex/bison 的语法分析器(C++语言实现)

C++略微麻烦一些，需要首先查看 ast.h 文件中对于 StmtAST 的数据结构定义，然后根据不同是 stmt 语句类型（sType）赋值即可，记得赋值和传参的时候使用 unique_ptr。代码略长只给出一小段：

```
IF LP Cond RP Stmt ELSE Stmt {
                $$ = new StmtAST();
                $$->selectStmt = unique_ptr<SelectStmtAST>(new SelectStmtAST());
                $$->sType = SEL;
                $$->selectStmt->ifStmt = unique_ptr<StmtAST>($5);
                $$->selectStmt->elseStmt = unique_ptr<StmtAST>($7);
                $$->selectStmt->cond = unique_ptr<LOrExpAST>($3);
            }
```

（3）基于 antlr 的语法分析器(C++语言实现)

①语法分析器

很快嗷，和 flex 只有语法上的区别，照着写就过了。代码如下：

```
stmt
    : lVal Assign exp Semicolon    # assign
    | exp? Semicolon    # exprStmt
    | block    # blockStmt
    | If Lparen cond Rparen stmt (Else stmt)?    # ifElse
    | While Lparen cond Rparen stmt    # while
    | Break Semicolon    # break
    | Continue Semicolon    # continue
    | Return exp? Semicolon # return
    ;
```

②AST 构造

也很快嗷，毕竟只需要写一个函数，稍微看一下 ctx 里面有啥，模仿其他代码写就过了，注意一下传参按照注释提示使用 move()函数传入右值。代码如下：

```
antlrcpp::Any AstVisitor::visitWhile(SysyParser::WhileContext *const ctx) {
    auto const cond_ = ctx->cond()->accept(this).as<Expression *>();
    std::unique_ptr<Expression> cond(cond_);
    auto const stmt_ = ctx->stmt()->accept(this).as<Statement *>();
```

```
        std::unique_ptr<Statement> stmt(stmt_);
        auto const ret = new While(std::move(cond), std::move(stmt));
        return static_cast<Statement *>(ret);
    }
```

4 中间代码生成

4.1 实验任务

在给出的中间代码生成器框架基础上完成 LLVM IR 中间代码的生成，将 Sysy 语言程序翻译成 LLVM IR 中间代码。

4.2 中间代码生成器的实现

文档很长嗷，但是不要怕，无论遇到什么样的困难，微笑着面对它。然后就会发现需要填写的只有一小段内容，根据文档中的编程要求。首先我们把 requireLval 设置为 true 表明这是一个赋值语句的左值，之后调用 Lval 的 accept() 就能正确处理；exp 同理，还需要在每个 accept 之后用变量保存 recentVal 的值方便后面使用，用 auto 自动推断类型即可；之后是两种需要进行类型转换的情况的处理->type_->tid_与 Type::IntegerTyID 和 Type::FloatTyID 进行对比（需要自行查看其他文件中相关数据结构的定义）；最后调用 create_store()函数进行赋值。代码如下：

```
case ASS: {
                // ****************** 代码填写处
                requireLVal = true;   //表明是赋值语句的左值
                ast.lVal->accept(*this);
                auto var = recentVal;
                is_single_exp = true;
                ast.exp->accept(*this);
                auto expval = recentVal;
                if (var->type_->tid_ == Type::FloatTyID && expval->type_->tid_ == Type::IntegerTyID) {
                    expval = builder->create_sitofp(expval,FLOAT_T);
                }
                else if (var->type_->tid_ == Type::IntegerTyID && expval->type_->tid_ == Type::FloatTyID) {
                    expval = builder->create_fptosi(expval, INT32_T);
                }
                builder->create_store(expval, var);
```

```
        // ****************** 代码结束
        break;
    }
```

5 目标代码生成

5.1 实验任务

在给出的代码框架基础上，将 LLVM IR 中间代码翻译成指定平台的目标代码：

(1) 基于 LLVM 的目标代码生成(ARM)

(2) 基于 LLVM 的目标代码生成(RISCV64)

以上任务任选一个完成即可。

5.2 目标代码生成器的实现

（1）基于 LLVM 的目标代码生成(ARM)

万事跟随实验文档来。首先初始化目标，几个函数已经给出，直接 ctrl+cv；选择 arm 平台的 target_triple 解除注释；最后就更简单了，注释部分给出了详细步骤，只需要注意语句顺序千万别错了就行。代码过长不在实验报告中展示了。

（2）基于 LLVM 的目标代码生成(RISCV64)

和 ARM 的几乎一样，改一下 target_triple 和 cpu 的值使其满足 riscv 平台即可。代码如下：

```
#include "codegen.h"
#include <memory>
#include <optional>
#include <string>

using namespace llvm;
using namespace llvm::sys;

namespace codegen {
std::string getGenFilename(const std::string &ir_filename,
                           const CodeGenFileType &gen_filetype) {
  if (gen_filetype == CGFT_Null) {
    return nullptr;
```

```
    }
    return ir_filename.substr(0, ir_filename.find(".")) +
                (gen_filetype == CGFT_AssemblyFile ? ".s" : ".o");
}

bool codeGenerate(const std::string &ir_filename,
                              const CodeGenFileType &gen_filetype) {
    SMDiagnostic error_smdiagnostic;
    LLVMContext context;
    std::unique_ptr<Module> module =
          parseIRFile(ir_filename, error_smdiagnostic, context);

    if (!module) {
        error_smdiagnostic.print(ir_filename.c_str(), errs());
        return false;
    }

    // Initialize the target registry etc.
    // ********************************************************************
    //  补充代码 1 - 初始化目标
    InitializeAllTargetInfos();
    InitializeAllTargets();
    InitializeAllTargetMCs();
    InitializeAllAsmParsers();
    InitializeAllAsmPrinters();




    //auto target_triple = module->getTargetTriple();
    //auto target_triple = getDefaultTargetTriple();
    auto target_triple = "riscv64-unknown-elf";
    //auto target_triple = "armv7-unknown-linux-gnueagihf";

    // ********************************************************************
    //  补充代码 2 - 指定目标平台


    module->setTargetTriple(target_triple);


    std::string error_string;
    auto target = TargetRegistry::lookupTarget(target_triple, error_string);

    // Print an error and exit if we couldn't find the requested target.
    // This generally occurs if we've forgotten to initialise the
    // TargetRegistry or we have a bogus target triple.
    if (!target) {
```

```
        errs() << error_string;
        return 1;
    }

    auto cpu = "generic-rv64";
    // auto cpu = "";
    auto features = "";

    TargetOptions opt;
    auto RM = Optional<Reloc::Model>();
    auto TheTargetMachine =
          target->createTargetMachine(target_triple, cpu, features, opt, RM);

    module->setDataLayout(TheTargetMachine->createDataLayout());

    //
**********************************************************************
    // 补充代码 3 - 初始化 addPassesToEmitFile()的参数，请按以下顺序
    // (1) 调用 getGenFilename()函数，获得要写入的目标代码文件名 filename
    // (2) 实例化 raw_fd_ostream 类的对象 dest。构造函数：
    //          raw_fd_ostream(StringRef Filename, std::error_code &EC, sys::fs::OpenFlags
Flags);
    //      Flags 置 sys::fs::OF_None
    //      注意 EC 是一个 std::error_code 类型的对象，你需要事先声明 EC，
    //      通常还应在调用函数后检查 EC，if (EC) 则表明有错误发生(无法创建目标文
件)，此时应该输出提示信息后 return 1
    // (3) 实例化 legacy::PassManager 类的对象 pass
    // (4) 为 file_type 赋初值。

    auto filename = getGenFilename(ir_filename, gen_filetype);
    std::error_code EC;
    raw_fd_ostream dest(filename, EC, sys::fs::OF_None);
    if(EC) return 1;
    legacy::PassManager pass;
    auto file_type = gen_filetype;

    if (TheTargetMachine->addPassesToEmitFile(pass, dest, nullptr, file_type)) {
        errs() << "TheTargetMachine can't emit a file of this type";
        return 1;
    }

    pass.run(*module);
    dest.flush();
    return true;
}


} // namespace codegen
```

6 总结

6.1 实验感想

感觉此次实验的总体难度适中，实验文档编写的很细致，几乎不需要自己额外搜索信息学习，关卡也是循序渐进。在做实验的过程中，遇到问题在群里提问老师也很贴心地进行了回答。但是我完成的也有不足之处，比如词法分析部分采用了面向答案编程。唯一的遗憾是有一关超时了才想起来写，导致没有让先辈的头像出现在排行榜上。

6.2 实验总结与展望

通过本次实验，我更加深入地了解了可执行文件的生成过程，对代码优化技术有了全新的认知，也对编译的全流程更了解了。以前觉得写代码已经比较接近底层，但是经过这个课程才发现原来计算机执行代码之前还有这么多准备工作。总之我们需要广泛学习编程各个方面的知识，能够帮助我们越发地理解计算机的工作方式，从而写出高效的代码。