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：　CS2004

学　　号：　U202015396

姓　　名：　沈子旭

电　　话：　13523859506

邮　　箱：　zixushen@hust.edu.cn

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：沈子旭

日期：2023 年　6 月 24 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 -o def-test def-test.c alibaba.c -I. -DBILIBILI
```

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 -S -O2 -target armv7-linux-gnueabihf -march=armv7-a -mfpu=neon -mfloat-abi=hard -o bar.clang.arm.s bar.c
```

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 -O2 -march=armv7-a -march=armv7-a -mtune=cortex-a7 -mfpu=neon -mfloat-abi=hard iplusf.c -o iplusf.arm.s
# 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 -static iplusf.arm.s sylib.a -o iplusf.arm
# 用 qemu-arm 运行 iplusf.arm
qemu-arm iplusf.arm
```

④make 的使用

Make 是一种用来进行项目构建的工具，将命令行中的复杂操作写到一个 make 文件中，以进行直接调用。以下是代码和解释部分：

```
CXXFLAGS := -g -Wall -Werror
CPPFLAGS := -Iinclude
LDLIBS   :=
```

这部分代码定义了一些变量：

CXXFLAGS 变量设置为 -g -Wall -Werror，它包含了编译器标志。-g 生成调试

信息，-Wall 开启所有警告信息，-Werror 将警告视为错误，中止编译过程。

CPPFLAGS 变量设置为 -Iinclude，它包含了预处理器标志。-Iinclude 指定了头文件的搜索路径为名为 "include" 的目录。

LDLIBS 变量为空，没有指定任何特定的链接库。

```
all: helloworld
helloworld: helloworld.cc main.cc
    $(CXX) $(CXXFLAGS) $(CPPFLAGS) $^ $(LDLIBS) -o $@
clean:
    $(RM) helloworld
```

使用编译器命令以及相关的编译器标志、预处理器标志和链接器标志，将源文件编译并链接生成可执行文件 helloworld。

（2）Sysy 语言及其运行时库

由于 SysY 语言与 c 语言很相似，因此只需实现一个寻找最大利润的算法。思路如下：一遍遍历，维护一个目前最低股价和目前最大利润变量 min_price 和 max_profit，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[]) {
    int max_profit = 0;
    int min_price = prices[0];
    int len = N;
    int i = 0;
    while (i < len) {
        if (prices[i] < min_price) {
            min_price = prices[i];
```

```
        } else if (prices[i] - min_price > max_profit) {
            max_profit = prices[i] - min_price;
        }
        i=i+1;
    }
    return max_profit;
}
```

（3）arm 汇编

ARM 汇编代码实现了一个升序的冒泡排序算法。它使用寄存器操作、循环和条件分支来实现排序过程。详细结构和逻辑可见代码注释：

```
.text
.global bubblesort
.syntax unified
.thumb
.thumb_func
bubblesort:
    push {r4, r5, r6, r7, lr}      @ 保存寄存器的旧值到栈中
    mov a2, #0                     @ 初始化 a2 为 0，作为程序结束时的 a1 的值
    mov r3, a0                     @ 将数组首地址保存到 r3
    mov r4, #0                     @ 初始化 r4 为 0，作为外层循环计数器
    subs a1, a1, #1                @ a1 减去 1，得到循环次数

LOOPSTART:
    cmp a1, r4                     @ 比较 a1 和 r4 的值
    ble FUNCEND                    @ 如果 a1 小于等于 r4，跳转到 FUNCEND 结束函数

    mov r7, a1                     @ 将 a1 的值保存到 r7，用作内层循环次数的差值
    mov r5, #0                     @ 初始化 r5 为 0，作为内层循环计数器

LOOPJ:
    ldr r6, [r3]                   @ 加载 r3 指向的元素值到 r6
    ldr r7, [r3, #4]               @ 加载 r3 后面 4 字节处的元素值到 r7
    cmp r6, r7                     @ 比较 r6 和 r7 的值
    it gt                          @ 根据比较结果设置条件
    strgt r7, [r3]                 @ 如果 r6 大于 r7，将 r7 的值存储到 r3 指向的位置
    strlt r6, [r3, #4]             @ 如果 r6 小于 r7，将 r6 的值存储到 r3 后面 4 字节处的位置
    adds r3, r3, #4                @ r3 加上 4，指向下一个元素
```

```
    adds r5, r5, #1              @ 内层循环计数器加 1
    cmp r5, r7                   @ 比较 r5 和 r7 的值
    blt LOOPJ                    @ 如果 r5 小于 r7，跳转到 LOOPJ 循环

    adds r4, r4, #1              @ 外层循环计数器加 1
    b LOOPSTART                  @ 跳转到 LOOPSTART 循环

FUNCEND:
    mov a0, #0                   @ 将 a0 设置为 0
    pop {r4, r5, r6, r7, pc}     @ 从栈中恢复之前保存的寄存器值

    bx lr                        @ 返回到调用该函数的位置
```

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)

首先我们需要写出正确的正则表达式以匹配标识符。int 型变量首先写出对于八进制、十进制和十六进制数字变量的识别表达式，用或语句连接；在数字前可能出现+-运算符，用[]表示任选其一；？出现或者未出现。*为 0 或更高词，+为 1 或更高次。float 型同理。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：

```
{FLOAT} {printf("%s : FLOAT_LIT\n", yytext); return FLOAT_LIT; }

{IDEN} {printf("%s : ID\n", yytext); return ID; }
```

```
{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;
}

{MultilineComment} {}
{SingleLineComment} {}

[ \n] {}
[ \r\t] {}
[ \n\t]+ {}
. {printf("Lexical error - line %d : %s\n",yylineno,yytext);return LEX_ERR;}
{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）

同上。

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

根据代码注释写出代码。(此处感谢杨老师的课堂指导 orz)

②语法分析

在语法检查的基础上，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 {$$ = new_node(Stmt, NULL, $1, $3, AssignStmt, 0, NULL, NonType);}
    | Exp SEMICOLON {$$ = new_node(Stmt, NULL, NULL, $1, ExpStmt, 0, NULL, NonType);}
    | SEMICOLON {$$ = new_node(Stmt, NULL, NULL, NULL, BlankStmt, 0, NULL, NonType);}
    | Block {$$ = new_node(Stmt, NULL, NULL, $1, Block, 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++相较于 C 语言部分麻烦一些，首先理解 ast.h 文件中对于 StmtAST 的数据结构定义，然后根据不同的 stmt 语句类型（sType）赋值即可。以下是部分代码块，逻辑和组成模式都相同：

```
WHILE LP Cond RP Stmt {
        $$ = new StmtAST();
        $$->sType = ITER;
        $$->iterationStmt = unique_ptr<IterationStmtAST>(new IterationStmtAST());
        $$->iterationStmt->cond = unique_ptr<LOrExpAST>($3);
        $$->iterationStmt->stmt = unique_ptr<StmtAST>($5);
    }
|
    BREAK SEMICOLON {
        $$ = new StmtAST();
        $$->sType = BRE;
    }
```

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，代码很短。首先把 requireLval 设置为 true 表明这是可赋值语句的左值，之后调用 Lval 的 accept 函数即可；exp 部分同理，在每个 accept 函数之后用变量保存 recentVal 的值以供后面使用，用 auto 函数自动推断类型；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，查看文档中对这两种类型的定义即可；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)

首先初始化目标，根据文档中的定义直接使用；选择 arm 平台的 target_triple 解除注释；根据注释部分直接编写代码，详情可见代码及注释。代码如下：

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

using namespace llvm;
using namespace llvm::sys;

namespace codegen {
    // 返回值：是否成功生成代码
    bool codeGenerate(const std::string &ir_filename, const CodeGenFileType &gen_filetype) {
        SMDiagnostic error_smdiagnostic;
        LLVMContext context;

        // 解析 IR 文件，生成 LLVM 模块
        std::unique_ptr<Module> module = parseIRFile(ir_filename, error_smdiagnostic, context);

        if (!module) {
```

```
        // 若解析失败，打印错误信息，并返回失败
        error_smdiagnostic.print(ir_filename.c_str(), errs());
        return false;
    }

    // 初始化目标平台相关信息
    InitializeAllTargetInfos();
    InitializeAllTargets();
    InitializeAllTargetMCs();
    InitializeAllAsmParsers();
    InitializeAllAsmPrinters();

    // 设置目标三元组为 "armv7-unknown-linux-gnueagihf"
    auto target_triple = "armv7-unknown-linux-gnueagihf";
    module->setTargetTriple(target_triple);

    std::string error_string;
    // 根据目标三元组查找目标平台
    auto target = TargetRegistry::lookupTarget(target_triple, error_string);

    if (!target) {
        // 若查找失败，打印错误信息，并返回失败
        errs() << error_string;
        return false;
    }

    auto cpu = "generic";
    auto features = "";

    TargetOptions opt;
    auto RM = Optional<Reloc::Model>();
    // 创建目标机器实例
    auto TheTargetMachine = target->createTargetMachine(target_triple, cpu, features,
opt, RM);

    // 设置模块的数据布局
    module->setDataLayout(TheTargetMachine->createDataLayout());

    // 获取生成的文件名
    auto filename = getGenFilename(ir_filename, gen_filetype);
    std::error_code EC;
    // 创建输出流
    raw_fd_ostream dest(filename, EC, sys::fs::OF_None);
    if (EC) return false;
    legacy::PassManager pass;
    auto file_type = CGFT_AssemblyFile;

    if (TheTargetMachine->addPassesToEmitFile(pass, dest, nullptr, file_type)) {
```

```
            // 若目标机器无法生成该类型的文件，打印错误信息，并返回失败
            errs() << "TheTargetMachine can't emit a file of this type";
            return false;
        }

        // 运行 PassManager，生成代码
        pass.run(*module);
        dest.flush();
        return true;
    }

    // 返回值：生成的文件名
    std::string getGenFilename(const std::string &ir_filename, const CodeGenFileType &gen_filetype) {
        if (gen_filetype == CGFT_Null) return nullptr;
        // 根据 IR 文件名和生成的文件类型，构造生成的文件名
        return ir_filename.substr(0, ir_filename.find(".")) + (gen_filetype == CGFT_AssemblyFile ? ".s" : ".o");
    }
}
```

6 总结

6.1 实验感想

本次实验难度看起来高，实则按照文档写就好。总体关卡难度适中，课程设计对经常去上课的同学非常友好（此处再次感谢杨老师的课堂指导）。在课程群中的提问也会被老师及时回答。总体来说，编译原理实验是诸多课程实验中最人性化的，感谢课程老师。

6.2 实验总结与展望

通过本次实验，我深入了解了可执行文件的生成过程，拓宽了对代码优化技术的认知，并对编译的全流程有了更深入的了解。之前我认为编写代码已经很接近底层了，但通过这个课程，我才意识到在计算机执行代码之前还需要进行大量的准备工作。这次实验让我意识到，我们需要广泛学习编程各个方面的知识，这有助于我们更好地理解计算机的工作方式，从而编写出更高效的代码。

了解可执行文件的生成过程让我明白了编译器在将源代码转换为可执行文件时所做的工作。这个过程包括词法分析、语法分析、语义分析、代码生成和优化等环节。编译器通过对代码进行分析和优化，可以提高程序的执行效率和性能。

对于代码优化技术，我认识到它们可以通过改变代码的结构、使用更高效的算法和数据结构、减少资源的使用等方式来提高程序的运行效率。优化技术可以使程序更快速、更节省资源，从而提升用户体验和系统性能。

同时，我也了解到编译的全流程并不仅仅是代码的转换和优化，还包括目标平台的选择、数据布局的设置以及生成可执行文件等环节。这些步骤对于确保程序在目标平台上正确运行至关重要。

综上所述，我们需要广泛学习编程的各个方面，不仅仅局限于写出能够实现功能的代码，还要了解计算机的工作方式、编译器的工作原理以及代码优化的技术，这样才能编写出高效、可靠的代码。这种全面的知识和理解将帮助我们更好地应对编程挑战，并为构建高性能的软件和系统打下坚实的基础。