图标

描述已自动生成

编译原理实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| **专业** | **：** |  |  |
| **班级** | **：** |  |  |
| **学号** | **：** |  |  |
| **姓名** | **：** |  |  |
| **电话** | **：** |  |  |
| **邮箱** | **：** |  |  |

**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：

|  |  |
| --- | --- |
| 综合成绩 |  |
| 教师签名 |  |

目 录

[1 编译工具链的使用 1](#_Toc136605994)

[1.1 实验任务 1](#_Toc136605995)

[1.2 实验实现 1](#_Toc136605996)

[2 词法分析 5](#_Toc136605997)

[2.1 实验任务 5](#_Toc136605998)

[2.2 词法分析器的实现 5](#_Toc136605999)

[3 语法分析 6](#_Toc136606000)

[3.1 实验任务 6](#_Toc136606001)

[3.2 语法分析器的实现 6](#_Toc136606002)

[4 中间代码生成 8](#_Toc136606003)

[4.1 实验任务 8](#_Toc136606004)

[4.2 中间代码生成器的实现 8](#_Toc136606005)

[5 目标代码生成 9](#_Toc136606006)

[5.1 实验任务 9](#_Toc136606007)

[5.2 目标代码生成器的实现 9](#_Toc136606008)

[6 总结 11](#_Toc136606009)

[6.1 实验感想 11](#_Toc136606010)

[6.2 实验总结与展望 11](#_Toc136606011)

# 编译工具链的使用

## 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。

## 实验实现

1. 编译工具链的使用
2. GCC编译器的使用

任务：用gcc编译器编译def-test.c和alibaba.c, 并指定合适的编译选项，生成二进制可执行代码def-test。执行的结果应当包括Bilibili的自我介绍以及Alibaba对BiliBili的喊话

GCC中用到的命令行选项有：

-o file 指定输出的文件为file。如果未指定该选项，在Linux下生成的可执行代码将被命名为缺省值a.out。指定输出文件名时，应使用缺省的后缀：预处理后：.i； 汇编代码：.s；目标代码.o等。

-DSOMETHING 宏定义,即预处理语句“#define SOMETHING”中的宏SOMETHING。

代码如下：

gcc def-test.c alibaba.c -o def-test -D BILIBILI

1. CLANG编译器的使用

任务：用clang编译器把程序bar.c“翻译”成优化的(优化级别O2)armv7架构，linux系统，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

Clang的命令行编译选项基本继承了gcc的大部分编译选项，如：-S, -o, -O, -c等。它也有一些gcc不支持的编译选项，比如交叉编译，通过指定-target参数，可以在X86的平台将C源程序“翻译”成其它平台下的汇编代码或二进制代码。

用到的命令行选项有：

-S 完成编译但不执行汇编，产生汇编文件

-O 批定优化级别，如：-O2,-O3等

-o file 指定输出的文件为file

代码如下：

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

1. 交叉编译器arm-linux-gnueabihf-gcc和qemu-arm虚拟机、

任务：用交叉编译器arm-linux-gnueabihf-gcc将源程序“翻译”成arm汇编代码，再将汇编代码汇编并与Sysy2022运行时库连接，生成arm可执行代码，然后用qemu-arm虚拟机运行arm可执行程序。

arm-linux-gnueabihf-gcc跟gcc的用法十分类似。在arm资源缺乏时，可以在X86架构的linux服务器上用qemu-arm虚拟机来运行arm的可执行代码，以检查测试用例是否被正确编译。使用的方法是：qemu-arm -L /usr/arm-linux-gnueabihf/ arm可执行代码

代码如下：

        arm-linux-gnueabihf-gcc -S iplusf.c -o iplusf.arm.s

        arm-linux-gnueabihf-gcc iplusf.arm.s sylib.a -o iplusf.arm

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

1. Make的使用

任务：编写一个 Makefile，使用 make 完成项目的构建。

项目结构如下：



图1 项目结构

Makefile文件由一系列规则构成。每条规则的形式如下：

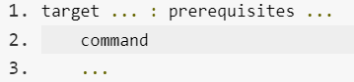


图2 makefile规则

Makefile变量使用：

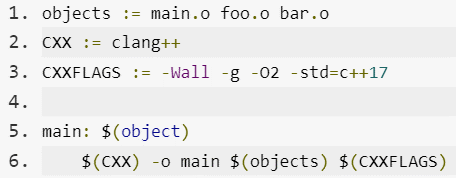


图3 makefile变量

使用 -I{path} 编译选项可以添加头文件路径

Makefile有三个非常有用的变量。分别是$@，$^，$<。

$@：目标文件，$^：所有的依赖文件，$<：第一个依赖文件。

代码如下：

        CXX = g++

        CXXFLAGS := -Wall -g -O2 -std=c++17

        INCLUDE = -I include

        SRC = main.cc helloworld.cc

        OBJ = $(SRC:.cc=.o)

        helloworld: $(OBJ)

        $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $(OBJ)

        %.o: %.cc include/helloworld.hh

        $(CXX) $(CXXFLAGS) $(INCLUDE) -c $< -o $@

1. Sysy语言及运行时库；

任务：熟悉SysY语言和运行时库，并用该语言写一个解决“买卖股票的最佳时机”的程序。

编程思路：遍历列表，更新最低股价和目前最大利润变量，得到过程中的最大利润。

代码如下：

        int max=0;

        int i=0,j;

        while(i<N-1){

         j=i+1;

         while(j<N){

if(prices[i]<prices[j]){

int tmp=prices[j]-prices[i];

if(tmp>max){

max=tmp;

}

}

j=j+1;

}

i=i+1;

}

return max;

1. 目标平台riscv64的汇编语言；

任务：用RISCV 汇编编写一个对数组排序的函数。

编程思路：泡沫排序，两层循环遍历交换，不再赘述

代码如下：

bubblesort:

mv s2, a0

addi s6, s2, 40

addi s7, s2, 36

loop1:

addi s3, s2 ,4

loop2:

lw s4, 0(s2)

lw s5, 0(s3)

bgt s4, s5, gg

j ll

gg:

sw s5, 0(s2)

sw s4, 0(s3)

ll:

addi s3, s3, 4

blt s3, s6, loop2

addi s2, s2, 4

blt s2, s7, loop1

# 词法分析

## 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。

## 词法分析器的实现

基于flex的Sysy词法分析器(C++实现)

任务：利用 flex 工具生成SysY2022语言的词法分析器，要求输入一个SysY2022语言源程序文件,比如test.c，词法分析器能输出该程序的token以及token的种别。

首先我们需要编写int、float和id的正则式，以及测试样例中的错误类型的正则式。int型分为10进制，8进制和16进制数字。float型分为科学计数法和不用科学计数法的。

代码如下：

DIGIT [0-9]

LETTER [A-Za-z]

int\_lit [1-9][0-9]\*|0[0-7]\*|(0x|0X)[0-9a-fA-F]\*

float\_lit [0-9]\*?\.?[0-9]\*[Ee][+-]?[0-9]\*f?|[0-9]\*?\.[0-9]\*f?

id [a-zA-Z\_][a-zA-Z0-9\_]\*

err 0[0-7]\*?[8-9][0-7]\*?|[0-9]\*[A-Za-z]\*[0-9]\*?

然后编写对识别出的token的规则：

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

{int\_lit} {printf("%s : INT\_LIT\n", yytext);return INT\_LIT;}

{float\_lit} {printf("%s : FLOAT\_LIT\n", yytext);return FLOAT\_LIT;}

{err} {printf("Lexical error - line %d : %s\n",yylineno,yytext);return LEX\_ERR;}

# 语法分析

## 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。

## 语法分析器的实现

1. 基于flex/bison的语法分析器(C实现)

任务：利用flex+bison生成SysY2022的语法分析程序。要求任给一个SysY2022语言的源程序，输出其抽象语法树(AST)。

实验文档已经写得很通俗易懂了。仿照其他现有代码，根据不同语句的语法规则，调用new\_node函数构造语法树结点。

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);};

# 中间代码生成

## 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。

## 中间代码生成器的实现

任务：生成LLVM IR中间代码

实验思路：

1. requireLval设置为true，表明这是一个赋值语句的左值，调用Lval的accept()，保存recentVal的值。
2. exp设置为true，调用Lval的accept()，保存recentVal的值。
3. 将第一个保存的recentVal->type\_->tid\_与Type::IntegerTyID和Type::FloatTyID进行对比，确定变量类型。
4. 调用create\_store函数进行赋值。

代码如下：

requireLVal = true;

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);

# 目标代码生成

## 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。

## 目标代码生成器的实现

基于LLVM的目标代码生成(RISCV64)

任务：使用 C++ 语言，将上一实验生成的LLVM IR中间代码翻译成riscv64的目标代码。允许调用LLVM的库函数实现上述功能。

1. 初始化目标

Copy实验文档里的代码就可以了

InitializeAllTargetInfos();

InitializeAllTargets();

InitializeAllTargetMCs();

InitializeAllAsmParsers();

InitializeAllAsmPrinters();

1. 指定目标平台

把target\_triple改为riscv64-unknown-elf

把cpu改为generic-rv64

1. 初始化addPassesToEmitFile()的参数，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置为sys::fs::OF\_None。EC是一个std::error\_code类型的对象，如果EC不为空，则说明目标文件创建出错，直接返回1.

(3) 实例化legacy::PassManager类的对象pass

(4) 为file\_type赋初值。

代码如下：

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;

}

# 总结

## 实验感想

实验不算很难，但也有一定的挑战性。实验文档编写的很详细且通俗易懂，提供了很多实验过程中需要前置知识和课外资料，只要愿意花时间认真读完基本就能顺利完成实验。在实验过程中，遇到问题在群里提问老师也很耐心地解答了。我觉得这门实验收获颇多。

## 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，对代码优化有了全新的认知。实验的内容也有很多是课堂上的理论知识没有涉及到的，比如SysY2022语言、操作系统架构和一些底层库函数的使用，拓宽了我对编译原理这门课知识的理解和应用。