图标

描述已自动生成

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](#_Toc136783329)

[1.1 实验任务 1](#_Toc136783330)

[1.2 实验实现 1](#_Toc136783331)

[2 词法分析 5](#_Toc136783332)

[2.1 实验任务 5](#_Toc136783333)

[2.2 词法分析器的实现 5](#_Toc136783334)

[3 语法分析 7](#_Toc136783335)

[3.1 实验任务 7](#_Toc136783336)

[3.2 语法分析器的实现 7](#_Toc136783337)

[4 中间代码生成 10](#_Toc136783338)

[4.1 实验任务 10](#_Toc136783339)

[4.2 中间代码生成器的实现 10](#_Toc136783340)

[5 目标代码生成 12](#_Toc136783341)

[5.1 实验任务 12](#_Toc136783342)

[5.2 目标代码生成器的实现 12](#_Toc136783343)

[6 总结 15](#_Toc136783344)

[6.1 实验感想 15](#_Toc136783345)

[6.2 实验总结与展望 15](#_Toc136783346)

# 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

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

# 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

（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;

}

}

# 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);

}

# 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

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;

}

# 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

（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

# 总结

## 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。

## 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。