图标

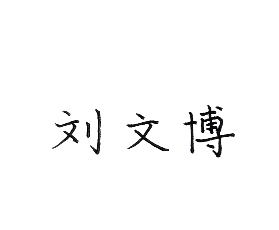
描述已自动生成

编译原理实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| **专业** | **：** | **计算机科学与技术** |  |
| **班级** | **：** | **BSB2101** |  |
| **学号** | **：** | **U202115666** |  |
| **姓名** | **：** | **刘文博** |  |
| **电话** | **：** | **15527700612** |  |
| **邮箱** | **：** | [**3097365481@qq.com**](mailto:3097365481@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：2023年 月 日

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

目 录

[1 编译工具链的使用 1](#_Toc169809488)

[1.1 实验任务 1](#_Toc169809489)

[1.2 实验实现 1](#_Toc169809490)

[2 词法分析 5](#_Toc169809491)

[2.1 实验任务 5](#_Toc169809492)

[2.2 词法分析器的设计与实现 5](#_Toc169809493)

[3 语法分析 10](#_Toc169809494)

[3.1 实验任务 10](#_Toc169809495)

[3.2 语法分析器的设计与实现 10](#_Toc169809496)

[4 静态语义分析 13](#_Toc169809497)

[4.1 实验任务 13](#_Toc169809498)

[4.2 静态语义分析的设计与实现 13](#_Toc169809499)

[5 中间代码生成 16](#_Toc169809500)

[5.1 实验任务 16](#_Toc169809501)

[5.2 中间代码生成器的实现 16](#_Toc169809502)

[6 目标代码生成 18](#_Toc169809503)

[6.1 实验任务 18](#_Toc169809504)

[6.2 目标代码生成器的设计与实现 18](#_Toc169809505)

[7 总结 19](#_Toc169809506)

[7.1 实验感想 19](#_Toc169809507)

[7.2 实验总结与展望 19](#_Toc169809508)

# 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

本节实验只需要掌握编译工具的使用即可：

1. GCC编译器的使用

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

该命令指定使用gcc工具，以def-test.c和alibaba.c作为输入，def-test作为输出，指定编译时的宏为BILIBILI。

1. [CLANG编译器的使用](https://www.educoder.net/tasks/kgycpq9szmfx)

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

该命令指定使用clang工具，只编译为汇编代码，指定目标平台告诉 clang 编译器生成适用于 ARMv7 架构的、带有硬件浮点运算（HF）的 Linux 平台的代码，指定优化级别为 2，以bar.c作为输入，bar.clang.arm.s作为输出。

1. 交叉编译器arm-linux-gnueabihf-gcc和qemu-arm虚拟机的使用

 # 用arm-linux-gnueabihf-gcc 将iplusf.c编译成arm汇编代码iplusf.arm.s

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

# 用qemu-arm运行iplusf.arm

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

首先指定用交叉编译器arm-linux-gnueabihf-gcc作为工具，将iplusf.c编译成汇编代码iplusf.arm.s，之后将生成的汇编代码和sylib.a链接到一块生成名为iplusf.arm的二进制程序。

最后用qemu-arm工具运行iplusf.arm程序。

1. Make的使用

# 用arm-linux-gnueabihf-gcc 将iplusf.c编译成arm汇编代码iplusf.arm.s

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

helloworld: main.o helloworld.o

g++ -o helloworld main.o helloworld.o

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

g++ -o main.o -c main.cc -I./include $(CXXFLAGS)

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

g++ -o helloworld.o -c helloworld.cc -I./include $(CXXFLAGS)

这一关需要我们掌握项目构建工具make的使用。

首先定义编译时的一些选项CXXFLAGS，之后给出各个目标的依赖和构建规则。注意-I选项后面紧跟头文件的路径，使用$(CXXFLAGS)来代替前面定义的编译选项。

一开始的makefile误把main当作了最终的目标，在运行时就会报错undefined reference to \_main\_，后来才发现实验的测试命令为make helloworld; ./helloworld将构建目标修改为helloworld即可通过测试。

1. Sysy语言及其运行时库

这一关要求熟悉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。

const int N = 10;

int prices[N];

// 请完成函数maxProfit(),其输入为股价数组，输出为可获得的最大利润

int maxProfit(int prices[]){

    // ----------  开始

    int ans = 0;

    int i = 0;

    while(i < N - 1)

    {

        int j = i + 1;

        while(j < N)

        {

            if(prices[j] - prices[i] > ans) ans = prices[j] - prices[i];

            j = j + 1;

        }

        i  = i + 1;

    }

    return ans;

    // ----------- 结束

}

// main()接收连续N个交易日的股价输入并存入数组prices[],

// 接着调用maxProfit()求可能的最大利润，然后输出该值，并换行。

int main(){

    // 股价数组的输入：

    int i = 0;

    while(i < N)

    {

        prices[i] = getint();

        i = i + 1;

    }

    int best = maxProfit(prices);

    //结果输出：

    putint(best);

    putch(10);

    return 1;

}

为此我们需要用getint函数读入输入的每天的股票价格，用maxProfit函数对prices数据做计算得出利润最大值，而maxProfit函数的思路也很简单，遍历买入和卖出的时间点(SysY2022语言中没有for语句，故用while循环进行枚举)，找出二者股票价格差值的最大值作为ans并返回ans。最后用putint函数输出ans，并用putch()函数输出换行符。

1. [RISC-V 汇编](https://www.educoder.net/tasks/yl5fj3hkz7mi)

本关要求用RISCV 汇编编写一个对数组排序的函数。

        .text

        .align  1

        .globl  bubblesort

        .type   bubblesort, @function

bubblesort:

    li      t0, 10         # 初始时无序

    li      t1, 0          # 初始计数器

outer\_loop:

    # 循环内部开始

    addi    t2, t1, 1         # 将外部循环计数器复制给内部循环计数器 t2

    li      t3, 0

inner\_loop:

    # 计算数组元素地址

    add     t4, a0, t3

    # 加载相邻的两个元素

    lw      t5, 0(t4)

    lw      t6, 4(t4)

    # 如果前一个元素大于后一个元素，则交换它们

    bge     t6, t5, no\_swap

    sw      t6, 0(t4)

    sw      t5, 4(t4)

no\_swap:

    addi    t2, t2, 1      # 内部循环计数器 t2 ++

    addi    t3, t3, 4

    blt     t2, t0, inner\_loop # 内部循环条件判断

    # 否则继续外部循环

    addi    t1, t1, 1      # 外部循环计数器 t1++

    blt     t1, t0, outer\_loop  # 跳转到外部循环开头

    j       .L2

.L2:

        li      a0,0

        ret

        .size   bubblesort, .-bubblesort

用riscv汇编语言编写了一个冒泡排序算法对数组元素进行排序。冒泡排序的两个循环的计数器用t1，t2进行保存，t2为内层循环的计数器。

数组的首地址保存在了a0寄存器中，我们可以用a0+偏移的方式访问数组中的元素。偏移量用t3寄存器来保存，a0的值不能改变，故a0+t3的计算结果用t4寄存器保存。

每次内层循环将t4地址处的数据和相邻的数据加载到t5，t6寄存器中，如果前者大于后者就交换，并将内层循环计数器t2自增1，t3自增一个int数据的size，同时判断内层循环是否结束。

内层循环结束后跳转到外层循环，外层循环计数器t1自增1，之后比较是否外层循环结束。外层结束之后数组的内容就是有序的了，此时将返回值0加载到a0中然后执行ret指令。

# 词法分析

## 实验任务

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。

需要补充的内容有：

(1) 标识符ID;

(2) int型字面量INT\_LIT;

(3) float型字面量FLOAT\_LIT;

(4) 词法错误

等规则的描述,以及对应的动作代码。

## 词法分析器的设计与实现

Flex工具实现的词法分析器包含三个部分：辅助定义部分、规则部分以及用户子程序部分。

其中规则部分为一系列匹配模式和动作，模式一般使用正则表达式书写，动作部分为C代码。

为了实现词法分析器，我们需要在规则部分补充实验所要求的4个内容的正则表达式及其动作c代码。

首先查找SysY2022语言的定义，在其中找到ID，INT\_LIT，FLOAT\_LIT的定义，如下图 2‑1 Ident语言定义图 2‑1、图 2‑2、图 2‑3。

可以发现



图 2‑1 Ident语言定义

ID由一个字母+若干个{字母|数字|下划线}构成，为了方便后续使用，在辅助定义部分定义LETTER和DIGIT。

DIGIT [0-9]

LETTER [A-Za-z]

则ID可以用下面的正则表达式来表示，并打印出ID。

{LETTER}({LETTER}|{DIGIT}|\_)\* {printf("%s : ID\n", yytext); return ID; }

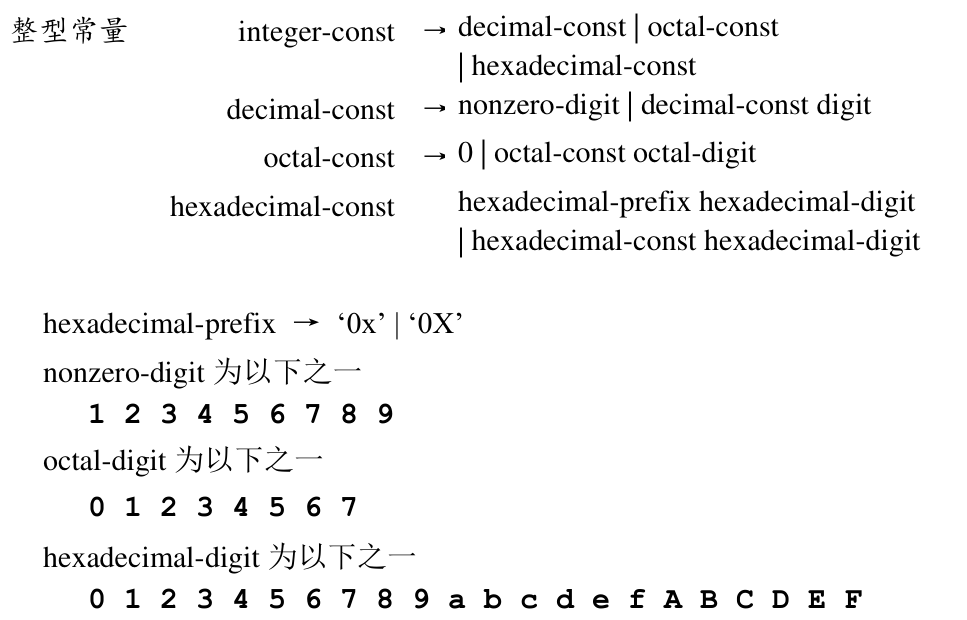


图 2‑2 整型常量的定义

整型变量既可以是10进制常量、8进制常量也可以是16进制常量。

故为了方便定义，在辅助定义部分引入如下定义

nonzero-digit [1-9]

octal-digit [0-7]

hexadecimal-digit [0-9a-fA-F]

hexadecimal-prefix 0[x|X]

possible-error-char [0-9a-zA-Z|\.]

decimal-const {nonzero-digit}{DIGIT}\*

octal-const 0|0[1-7]{octal-digit}\*

hexadecimal-const {hexadecimal-prefix}[1-9a-fA-F]{hexadecimal-digit}\*

integer-const {decimal-const}|{octal-const}|{hexadecimal-const}

其中10进制数由开头的一个非零数字加若干个DIGIT组成，8进制数由0或0开头的没有前导0的若干octal-digit组成，16进制数由0x或0X开头，加上没有前导0的若干hexadecimal-digit组成。

这些正则表达式严格参照SysY2022语言的定义进行设计。

最终整型常量可以用下面的正则表达式进行匹配：

{integer-const} {printf("%s : INT\_LIT\n", yytext); return INT\_LIT; }

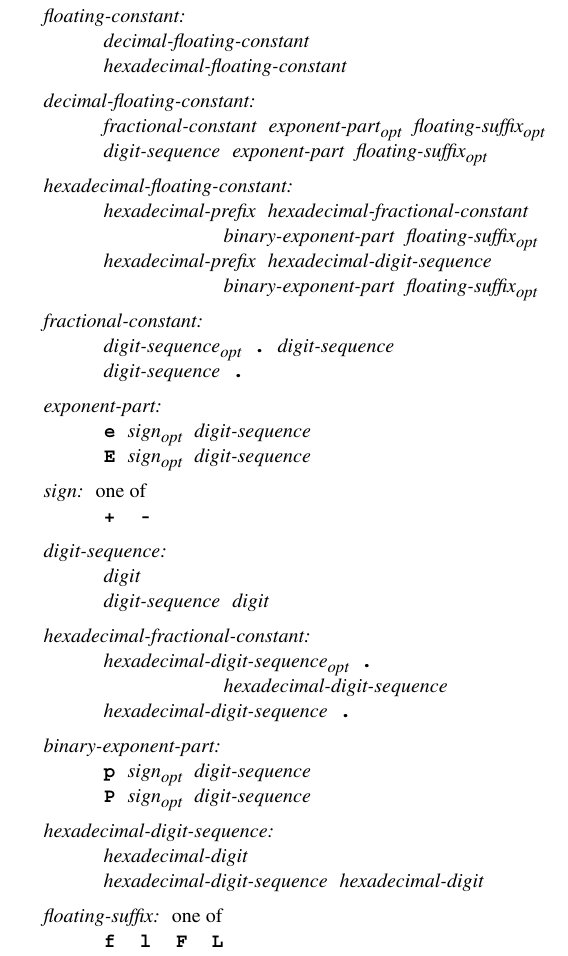


图 2‑3 浮点常量的语言定义

浮点常量的定义较为复杂，同样地严格参照SysY2022中浮点常量的定义，分模块地设计正则表达式。

浮点常量可以10进制的也可以是16进制的，无论是哪种进制都即可以是带有小数点的浮点数，也可以是有数字序列+[指数部分]+[浮点后缀]。

将这些规则写成正则表达式的形式，得到floating-constant的定义。

sign [+|-]

floating-suffix [f|l|F|L]

digit-sequence {DIGIT}+

hexadecimal-digit-sequence {hexadecimal-digit}+

exponent-part ([e|E]({sign})?{digit-sequence})

binary-exponent-part ([p|P]({sign})?{digit-sequence})

fractional-constant (({digit-sequence})?\.{digit-sequence})|({digit-sequence}\.)

hexadecimal-fractional-constant (({hexadecimal-digit-sequence})?\.{hexadecimal-digit-sequence})|({hexadecimal-digit-sequence}\.)

hexadecimal-floating-constant {hexadecimal-prefix}({hexadecimal-fractional-constant}|{hexadecimal-digit-sequence}){binary-exponent-part}({floating-suffix})?

decimal-floating-constant (({fractional-constant}({exponent-part})?)|({digit-sequence}{exponent-part}))({floating-suffix})?

floating-constant {decimal-floating-constant}|{hexadecimal-floating-constant}

则浮点常量可以用下面的语句进行匹配，匹配成功时打印FLOAT\_LIT。

{floating-constant} {printf("%s : FLOAT\_LIT\n", yytext); return FLOAT\_LIT; }

浮点常量的匹配语句位于整型常量之后。

在规则部分的最后（优先级最低，仅当上面的所有规则都不发生匹配时才触发），加上一条规则吸收所有的不匹配字符，并打印错误行号和错误文本。

{possible-error-char}+|. {printf("Lexical error - line %d : %s\n",yylineno,yytext);return LEX\_ERR;}

其中possible-error-char定义在辅助定义部分。

possible-error-char [0-9a-zA-Z|\.]

# 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 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。

## 语法分析器的设计与实现

本关需要在实验给出的parser.y的基础上，补充SysY语言(2022版)定义中的stmt语法部分：

Stmt → LVal '=' Exp ';' | [Exp] ';' | Block

| 'if' '( Cond ')' Stmt [ 'else' Stmt ]

| 'while' '(' Cond ')' Stmt

| 'break' ';' | 'continue' ';'

| 'return' [Exp] ';'

为了实现这个语法的扩展，我们首先需要在ast.h中查看StmtAST的数据结构定义：

class StmtAST : public BaseAST {

public:

    STYPE sType;

    unique\_ptr<LValAST> lVal = nullptr;

    unique\_ptr<AddExpAST> exp= nullptr;

    unique\_ptr<ReturnStmtAST> returnStmt = nullptr;

    unique\_ptr<SelectStmtAST> selectStmt = nullptr;

    unique\_ptr<IterationStmtAST> iterationStmt = nullptr;

    unique\_ptr<BlockAST> block = nullptr;

    void accept(Visitor &visitor) override;

};

可以看到StmtAST包含有sType，lVal，returnStmt,selectStme,iterationStmt,blok等成员，且除了sType其初值都是nullptr。我们在构建AST时需要根据不同的情况进行赋值。

以Stmt 推导成 LVal '=' Exp ';'为例：

首先写出Stmt在这种情况下的语法规则Stmt: LVal ASSIGN Exp SEMICOLON。匹配成功后，我们先创建一个StmtAST对象并赋值给$$符号(当前语法规则生成的值)，并尝试初始化它。

首先在define.h中找到STYPE的定义

enum STYPE {

    SEMI, ASS, EXP, CONT, BRE, RET, BLK, SEL, ITER

};

这里对应者赋值语句ASS，故将$$->sType 初始化为ASS。

根据StmtAST的成员，我们还需要给lVal和exp成员赋值，其类型分别为LValAST和AddExpAST，同样地去ast.h中查看LValAST和AddExpAST的数据结构定义。

class LValAST:public BaseAST {

public:

    unique\_ptr<string> id;

    vector<unique\_ptr<AddExpAST>> arrays;

    void accept(Visitor &visitor) override;

};

class AddExpAST : public BaseAST {

public:

    unique\_ptr<AddExpAST> addExp;

    unique\_ptr<MulExpAST> mulExp;

    AOP op;

    void accept(Visitor &visitor) override;

};

我们需要将当前规则中的第一个元素的值(即$1 LVal)赋值给lVal成员，将第三个元素的值(即$3 Exp)赋值给exp成员。

最终得到下面的语法规则：

Stmt:   LVal ASSIGN Exp SEMICOLON {

            $$ = new StmtAST();

            $$->sType = ASS;

            $$->lVal = unique\_ptr<LValAST>($1);

            $$->exp = unique\_ptr<AddExpAST>($3);

        }

虽然在这个例子中查看LValAST和AddExpAST的数据结构定义并没有什么用，但在Stmt -> 'if' '( Cond ')' Stmt [ 'else' Stmt ] 或者 Stmt -> 'while' '(' Cond ')' Stmt 这样的语法规则中，查看Stmt成员的数据结构定义就变得很有必要了。因为我们可能会需要将规则中的某个元素赋值给Stmt成员的成员(例如将'while' '(' Cond ')' Stmt中的Cond赋值给Stmt的iterationStmt成员的cond成员，如果不查看iterationStmt显然是无法做到这一点的)。

用上述的方法对所有情况进行分析，最终补全Stmt的语法规则即可。

值得注意的是在处理Stmt推导成'return' [Exp] ';'时，如果没有Exp，直接'return'';'，则不能简单的将returnStme置为nullptr，而必须创建一个ReturnStmtAST对象，将该对象的exp成员置为nullptr。否则，就会出现栈错误。这一点比较容易出错。

# 静态语义分析

## 实验任务

修改语义检查器Checker的源文件src/checker/checker.h和src/checker/checker.cpp，需要能检测出以下语义错误：

1. Redefined Variable 当前作用域重复定义变量，以及函数形参重复定义
2. Use Undefined Variable 使用未定义变量
3. Redefined Function 函数重复定义
4. Use Undefined Function 使用未定义函数
5. Can not Match Function Parameters 函数参数/类型不匹配，即函数调用必须保证实际参数的个数和类型都与函数声明中的形式参数完全匹配。
6. Func return type not match 函数返回值类型不匹配, 例如函数返回类型void/float时，函数内出现带返回值为int的return语句
7. Array index not int 数组下标不是整数
8. Break not in loop break语句不在循环中
9. Continue not in loop continue语句不在循环中
10. Visit non-array variable in the form of subscript variables 对非数组变量采用下标变量的形式访问

## 静态语义分析的设计与实现

该关要求我们在遍历语法树的过程中记录信息，进行静态语义分析检测出实验要求的10种错误。

该关为语义错误定义了枚举类ErrorType：

enum class ErrorType {

VarUnknown = 1,

VarDuplicated,

FuncUnknown,

FuncDuplicated,

FuncParamsNotMatch,

FuncReturnTypeNotMatch,

ArrayIndexNotInt,

BreakNotInLoop,

ContinueNotInLoop,

VisitVariableError,

};

分别代表了不同的错误类型，同时ErrorReporter提供了error方法，该方法接受两个参数：type表明出现的ErrorType，ast表示出错的节点的名称。

例如： 类FuncDefAST用于表示函数定义，包含一个属性id，表示函数的标识符，（int gcd(int a, int b) {...}中的gcd）,当检测到函数重复定义的语义错误时，调用error方法，并退出程序。

err.error(ErrorType::FuncDuplicated, \*node.id);

exit(int(ErrorType::FuncDuplicated));

类似的，当我们根据语义分析过程中记录的信息检测到错误时，也应调用error方法指明错误类型和错误节点名称(如果有的话)，并以对应的返回值退出。

下面给出不同语义错的检测和报告过程：

* 函数返回类型不匹配

err.error(ErrorType::FuncReturnTypeNotMatch, "");

exit(int(ErrorType::FuncReturnTypeNotMatch));

语义错类型为FuncReturnTypeNotMatch，由于returnStmAST没有id成员，故错误节点名称为空串。最后以FuncReturnTypeNotMatch为返回值退出。

* 左值数组下标不是整数

if (this->current\_type.type != TYPE\_INT) {

err.error(ErrorType::ArrayIndexNotInt, \*ast.id);

exit(int(ErrorType::ArrayIndexNotInt));

}

checker.h 中定义了 current\_type 变量记录当前类型，故只需要判断this->current\_type.type != TYPE\_INT即可检查出该语义错。

* 对非数组变量采用下标变量的形式访问

err.error(ErrorType::VisitVariableError, \*str);

exit(int(ErrorType::VisitVariableError));

前面已经执行了auto str = ast.id.get();得到了当前ast单元的id了，故error方法的后一个参数为\*str。

* break语句不在循环中

if (this->in\_loop == false) {

err.error(ErrorType::BreakNotInLoop, "");

exit(int(ErrorType::BreakNotInLoop));

}

checker.h中Checker类定义了in\_loop成员用来记录当前是否在循环中。

故break语句不在循环中语义错的检查逻辑就是判断this->in\_loop是否为false。如果是调用error方法报告错误。

但注意需要在StmtAST的visit函数中添加对iterationStmt的检查，否则会忽略这种情况，不对iterationStmt节点及其儿子节点做任何检查。自然就无法检测出break不再循环中的错误了。

if (ast.iterationStmt) {

ast.iterationStmt->accept(\*this);

}

* continue语句不在循环中

与break不再循环中的检查逻辑类似。

函数未定义、函数参数/类型不匹配的语义错的检查逻辑已经定义，只需要调用error方法报告错误即可，不再赘述。

# 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

该关要求补全visit(StmtAST &ast)方法，使其能够为赋值语句生成中间代码。

简单地説，需要作的工作是：

1. 在visit(lVal)之前通过全局临时变量，传递信息，表示当前的lVal是赋值语句左值，不是表达式。
2. 在visit(lVal)之后从recentVal取左值的Value(一个Value类的对象)。
3. 在visit(exp)后，从recentVal取右值的Value(也是一个Value类的对象).
4. 检查赋值语句左值和右值的类型，必要时作类型转换。LLVM IR是强类型语言，所有指令的两个操作数必须是同一类型。

比如：

当左值为指向i32的指针而右值是一float类型的值时，右值应当转为i32。

当左值为指向float的指针而右值是一个int类型的值时，右值应当转为float。

1. 调用IRStmtBuilder::create\_store(Value \*val, Value \*ptr)方法生成store指令。

为此，首先查看visit(LValAST &ast)的定义，发现该方法首先执行bool isTrueLVal = requireLVal; //是否真是作为左值，查看全局变量requireLVal

的值，进而判断是否是左值。如果是左值，则左值表达式的地址会保存到recentVal中。recentVal是一个Value类的指针。

之后在访问exp成员，得到赋值表达式右部的值，同样保存在recentVal中。

之后判断两个指针的类型是否一致，如果不一致进行类型转化。查看ir.h中Value的定义，发现explicit Value(Type\* ty, const std::string& name = "") : type\_(ty), name\_(name) {}，Value类具有成员type\_。type\_具有类型Type \*。

查看Type类的定义

class Type {

public:

    enum TypeID {

        VoidTyID,      // Void

        LabelTyID,     // Labels, e.g., BasicBlock

        IntegerTyID,   // Integers, include 32 bits and 1 bit

        FloatTyID,     // Floats, only 32 bits

        FunctionTyID,  // Functions

        ArrayTyID,     // Arrays

        PointerTyID,   // Pointer

    };

    explicit Type(TypeID tid) : tid\_(tid) {}

    ~Type() = default;

    virtual std::string print();

    TypeID tid\_;

};

发现，Type具有成员tid\_，tid显式地指出了类型。

故据此对两个Value \*指针的类型进行判断。

最后调用builder->create\_store方法生成store命令即可。

补全的代码如下所示：

requireLVal = true;

ast.lVal->accept(\*this);

Value \* address = recentVal;

is\_single\_exp = true;

ast.exp->accept(\*this);

Value \* data = recentVal;

if(address -> type\_ -> tid\_ == Type::IntegerTyID && data -> type\_ ->tid\_ == Type::FloatTyID) data = builder->create\_fptosi(data,INT32\_T);

if(address -> type\_ -> tid\_ == Type::FloatTyID && data -> type\_ ->tid\_ == Type::IntegerTyID) data = builder->create\_fptosi(data,FLOAT\_T);

builder->create\_store(data, address);

# 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。

## 目标代码生成器的设计与实现

本关要求根据上一节生成的中间代码，产生ARMv7平台的目标代码。

具体而言需要补充的代码共分三段，包括：

1. 初始化目标的registry;
2. 设置目标平台;
3. 初始化addPassesToEmitFile()函数的参数。

只需要参考实验文档一步一步补充代码即可。

从codegen.h中找到getGenFilename的定义，掌握其用法后，将正确的参数填入codegen.cc中获取要写入的目标代码文件名。

std::string getGenFilename(const std::string &ir\_filename,

                           const CodeGenFileType &gen\_filetype);

实验代码较长，且难度不大这里就不贴出来了。

# 总结

## 实验感想

编译原理实验总体而言做得很好，从实验文档的详尽程度上可以看出课程组的用心。做完实验也感觉对编译器的实现有了更进一步的认识。

但做完实验难免感觉前端和后端有点头重脚轻，中间代码生成和目标代码生成的部分有点一笔带过。这两关可以完全不看实验文档的其他部分，根据编程要求直接写。个人认为原因在于，编译原理实验作为一个可以直接运行的编译器项目，后端部分来到了项目的深水区，同学们实在难以掌握，所以干脆实验就布置的很简单。现有的中间代码生成和目标代码生成实验，感觉更多的是在教如何用工具或定义好的函数。之前在课余看了看斯坦福的编译原理课程，他们的代码生成模型则是一个1-register机（或者更进一步是一个提前计算需要多少个临时变量，给过程计算分配寄存器的模型）。课程的讲师在课上详细解释了1-register机的原理并推荐学生用这个模型去给COOL语言生成代码，并说这是一个低效但完全可行的想法，这对COOL语言来说足够了。个人认为，如果未来能够简化代码生成的任务，让同学们更多地思考其中的原理就更好力。

## 实验总结与展望

通过本次课程的一系列实验，我获得了以下收获：

1. 掌握了gcc、clang等编译工具和flex等编译前后端工具的使用。
2. 掌握了 Sysy 语言的定义及其运行时库的使用。
3. 学会了根据语言定义编写正则表达式匹配源程序
4. 能够使用flex根据产生式进行语法检查
5. 掌握了LLVM IR的生成流程和不同平台目标代码的生成流程

总的来说，编译原理实验给我带来了很多收获，锻炼了我的代码能力，加深了我对编译器相关知识的理解。

希望课程能够越办越好😄！