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：　U202015409

姓　　名：　李学森

电　　话：　18998023220

邮　　箱：　2575067251@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：李学森

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；

第 1 关：GCC 编译器的使用

在本关中，需要用 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。

由于文件中需要宏定义 BILIBILI 来展示 BILIBILI 的对话，因此需要使用-D BILIBILI 参数来实现宏定义的功能，又因为二进制可执行代码为 def-test，因此要使用 -o def-test 来指定输出的文件名字。

所以具体代码如下所示：

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

第 2 关：CLANG 编译器的使用

在本关中，需要使用 clang 编译器把上述程序 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（本程序并没有浮点数）。汇编代码文件名为 bar.clang.arm.s。

本关中需要使用-O2 参数来指定优化等级;-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;-S 参数表示生成汇编程序;-O 参数表示指定的输出文件名字。

所以具体代码如下所示：

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

第 3 关：交叉编译器 arm-linux-gnueabihf-gcc 和 qemu-arm 虚拟机的使用

本关中需要用 arm-linux-gnueabihf-gcc 将 iplusf.c 编译成 arm 汇编代码 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，最后用 qemu-arm 运行 iplusf.arm

本关难度较低，只需要通过仿照示例代码，修改相应的参数即可。

具体代码实现如下：

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

第 4 关：make 的使用

本关中，我们完成 Makefile，完成 helloworld 项目的构建，为 helloworld 目标编写一条生成一个名为 helloworld 的可执行文件的规则。

为了简化代码，在 Makefile 中，使用了 objects := helloworld.cc main.cc 变量语句，后续所有使用到 helloworld.cc main.cc 的地方均用 $(object)代替。同时由于头文件 helloworld.hh 存放在 include 文件夹中，需要使用-I 参数来指定头文件的路径。

具体实现代码如下：

```
objects := helloworld.cc main.cc

helloworld : $(object)
    g++ -o helloworld $(objects) -I ./include/
```

(2) Sysy 语言及其运行时库

第 1 关：Sysy 语言与运行时库

本关任务：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。程序要求给定一个数组 prices ，它的第 i 个元素 prices[i] 表示一支给定股票第 i 个交易日的价格(假定股价是整数)。你只能选择某个交易日买入这只股票，并选择在未来的另一个交易日卖出该股票。设计一个算法来计算你所能获取的最大利润，并返回这个最大利润值。如果你不能获取任何利润，返回 0 。

主要思路是 maxProfit 函数中，对数组 prices 进行遍历操作，在遍历的过程中维护最小价格，同时计算当前价格和最小价格的差值，若差值大于当前的结果变量 ans，则将差值赋值到 ans 中，最后返回 ans 到主函数中。

具体代码如下所示:

```
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 minPrice=prices[0];
    int i=1;
    int ans=0;
    while(i<N)
    {
        if(prices[i]<minPrice)
        {
            minPrice=prices[i];
        }
        else if(prices[i]-minPrice>ans)
        {
            ans=prices[i]-minPrice;
        }
        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;
}
```

(3) 目标平台 arm 的汇编语言

第 1 关：arm 汇编

本关的任务是使用 arm 汇编语言，完善 bubblesort 函数，以此实现冒泡排序。

Bubblesort 函数接受两个参数，数组 arr 的首地址保存在寄存器 r0 中，数组元素的个数 n 保存在寄存器 r1 中。该函数首先使用 push 指令将寄存器 r2 到 r8 和 lr 保存到堆栈中。然后它将 r2 初始化为 0，并使用公式 4*(r1-1)计算最后一个元素在数组中的字节偏移量，并将其存储在 r8 中。然后计算数组最后一个元素的地址并将其存储在 r6 中。

该算法由两个嵌套循环组成。外部循环迭代 r1 - 1 次，内部循环比较数组的相邻元素，并在它们的顺序不正确时交换它们。如果在迭代中没有进行交换，则内部循环提前退出。在每次外部循环迭代结束时，通过将 r6 减去 4 来在下一次迭代中排除数组的最后一个元素。

排序完成后，函数使用 pop 指令从堆栈中恢复寄存器，并使用 bx lr

指令返回到调用者。

具体代码实现如下：

```
    .arch armv7-a
    .text
    .global bubblesort
    .syntax unified
    .thumb
    .thumb_func
bubblesort:
    push {r2-r8,lr}
    mov r2, #0
    mov r8, #4
    mul r8, r8, r1
    sub r8 , #4

    mov r6, r0
    add r6, r6, r8
outer_loop:
    mov r3, r0
inner_loop:
    ldr r4, [r3]
    ldr r5, [r3, #4]
    cmp r4, r5
    ble no_swap
    str r4, [r3, #4]
    str r5, [r3]

no_swap:
    add r3, r3,#4
    cmp r3, r6
    blt inner_loop

    add r2, r2, #4
    cmp r2, r8
    bgt end

    sub r6, r6, #4
    b outer_loop
end:
    mov r0, #0
    pop {r2-r8,pc}
    bx lr
```

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)

第 1 关：用 flex 生成 SysY2022 语言的词法分析器

本关要求在文件 sysy.l 中补充适当的代码，设计识别 SysY2022 语言单词符号的词法分析器。需要补充的内容有：①标识符 ID;②int 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。

工作总体分为两步，首先通过编写正则表达式来识别对应的语法规则，然后对相应规则进行动作（打印当前识别的字符串的语法类型以及返回其对应的 token 值）。

使用到的正则表达式的规则如下：

①[]表示任选其一。

②? 表示出现或者不出现。

③*表示出现 0 次及以上。

④+表示出现 1 次及以上。

⑤-两端连接字母或者数字，表示使用这两个字母或数字中间的所有数。

原来由题目可知，只需要分别写出标识符、int 型以及 float 型变量的正则式并附上对应动作即可，剩余的便属于语法规则错误。后面发现 int 型

变量中的 16 进制数，开头为 0x 或者 0X，属于数字加字母开头，与词法错误容易混淆，因此将 16 进制整数的识别提前，单独识别，然后接着进行词法错误识别。最后进行标识符，8/10 进制整数识别以及浮点数识别。

具体代码实现如下:

```
0[xX][A-Fa-f1-9][A-Fa-f0-9]* {printf("%s : INT_LIT\n",
yytext); return INT_LIT;}
0[8-9]+[0-9]*|[0-9]+[fF]|[0-9]+[A-Z_a-z]+[A-Z_a-z0-9]*
{printf("Lexical error -
line %d : %s\n",yylineno,yytext);return LEX_ERR;}
[A-Z_a-z][A-Z_a-z0-9]* {printf("%s : ID\n", yytext); return
ID;}
[1-9][0-9]*|0[1-7][0-7]*|0 {printf("%s : INT_LIT\n", yytext);
return INT_LIT;}
[0-9]*(\.[0-9]+)?[eE][-+]?[0-9]+[fF]?|[0-9]*\.[0-9]+[fF]?
{printf("%s : FLOAT_LIT\n", yytext); return FLOAT_LIT;}
```

（2）基于 flex 的 Sysy 词法分析器(C++实现)

第 1 关：用 flex 生成 SysY2022 语言的词法分析器

本关要求与（1）基于 flex 的 Sysy 词法分析器(C 语言实现)相同，且对于①标识符 ID;②int 型字面量 INT_LIT;③ float 型字面量 FLOAT_LIT;④词法错误这四点的规则，C++与 C 语言相同，因此本关代码与（1）中完全相同。具体代码参见（1）。

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)

第 1 关:Sysy2022 语法检查

本关要求完善 parser.y 的语法规则和语义计算规则,实现语法检查和语法分析。只要正确写出文法的产生式(配上语义动作:{$$ = NULL;})即可通过语法检查。正确写出创建 AST 的语义动作才能完成语法分析,并生成 AST。

由于在代码注释中,给出了我们需要完善的语法规则和语义计算规则的框架,我们只需要根据注释进行相应的替换即可。

具体代码如下:

```
Stmt : LVal ASSIGN Exp SEMICOLON {$$ = NULL;}
     | Exp SEMICOLON {$$ = NULL;}
     | SEMICOLON {$$ = NULL;}
     | Block {$$ = NULL;}
     | IF LP Cond RP Stmt ELSE Stmt {$$ = NULL;}
     | IF LP Cond RP Stmt {$$ = NULL;}
     | WHILE LP Cond RP Stmt {$$ = NULL;}
     | BREAK SEMICOLON {$$ = NULL;}
     | CONTINUE SEMICOLON {$$ = NULL;}
     | RETURN Exp SEMICOLON {$$ = NULL;}
     | RETURN SEMICOLON {$$ = NULL;}
```

第 2 关:SysY2022 语法分析

本关要求我们在第一关的基础上,将语义动作补充完整,使用 new_node

函数构建抽象语法树的子节点，帮助其顺利完成 AST 的构建即可。本题需要注意的是传参的顺序以及语句类型的选择，函数使用参考其他部分的代码以及实验的参考文档。

创建 AST 子树时，由于所有节点共用一个函数，不是每个字段对每类节点都有意义，对不需要的字段分别置为 Null, 0, NonType 等即可.1 个子节点用 right， 两个用 left,right，三个用 left,mid,right。

具体代码实现如下：

```
Stmt : LVal ASSIGN Exp SEMICOLON {$$ =
new_node(Stmt,$1,NULL,$3,AssignStmt,0,NULL,NonType);}
     | Exp SEMICOLON {$$ =
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);}
```

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

(1) 中间代码生成 (C++语言实现))

第一关：生成 LLVM IR 中间代码

本关要求我们在 genIR.cpp 文件中，实现 genIR.h 中说明的 visit()方法(genIR.cpp 的 410 行之后) void visit(StmtAST &ast) override; 中，赋值语句的翻译。

根据文档所说，我们的工作总体来说分为以下 4 步:

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。

首先，我们将 requireLVal 设为 true，表明这是一个赋值语句的左值。

然后定义 Value *类型的长度为 2 的数组 vals 用于接受数据。之后调用 ast.lVal->accept(*this)，然后将 recentVal 存放到 vals[0]中；同理，调用 ast.exp->accept(*this)，然后将 recentVal 存放到 vals[1]中。然后检查赋值语句左值和右值的类型，若类型不相同则调用 builder->create_fptosi() 函数进行强制类型转换。最后使用 builder->create_store()函数进行赋值操作。

具体代码实现如下：

```
case ASS: {
            // ****************** 代码填写处
            requireLVal=true;
            Value *vals[2];
            ast.lVal->accept(*this);
            vals[0]=recentVal;
            ast.exp->accept(*this);
            vals[1]=recentVal;
            if(vals[0]->type_==INT32_T &&
vals[1]->type_==FLOAT_T)
            {
                vals[1]=builder->create_fptosi(vals[1],INT32_T);
            }else if(vals[0]->type_==FLOAT_T &&
vals[1]->type_==INT32_T)
            {
                vals[1]=builder->create_fptosi(vals[1],FLOAT_T);
            }
            builder->create_store(vals[1],vals[0]);
            // ****************** 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)

第 1 关：使用 LLVM 工具链生成目标代码(ARMv7)

本关任务是使用 C++ 语言，将上一实验生成的 LLVM IR 中间代码翻译成 ARM 的目标代码。

根据题目以及代码注释，本关分为三步：

①初始化目标

这一步只需要根据参考文档出的函数顺序直接填入代码，然后根据目标代码所在的平台，取消对应的注释即可。

具体代码如下：

```
InitializeAllTargetInfos();
InitializeAllTargets();
InitializeAllTargetMCs();
InitializeAllAsmParsers();
InitializeAllAsmPrinters();
```

②指定目标平台

由于本关的目标代码为 ARMv7，因此需要在注释中解除该段代码的注释，以此成功指定目标代码的平台。

具体代码如下：

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

③初始化 addPassesToEmitFile()的参数

本步骤所需要完成的具体操作都已经写在注释中，只需要根据注释逐步完善即可。

具体代码如下所示：

```
// (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。
  std::string
Filename=getGenFilename(ir_filename,gen_filetype);
  std::error_code EC;
  raw_fd_ostream dest(Filename, EC, sys::fs::OF_None);
  if(EC)
  {
    errs() << "Error happen, can't create target file";
    return 1;
  }
  legacy::PassManager pass;
  CodeGenFileType file_type=gen_filetype;
```

6 总结

6.1 实验感想

本次实验的总体难度不大。实验让我们先从文件链接编译对编译有初步的认识，然后逐步深入，从词法分析，语法分析，中间代码生成，目标代码生成这几个重要环节对编译有更加深刻的认识。由于实验文档非常的详细，还有实验讲解视频的辅助，基本上不需要我们去额外搜索资料，大大节约了我们的时间。在一些关键的实验步骤，老师还会附上详细的步骤注释，同样大大降低了实验的难度。

6.2 实验总结与展望

本次实验中，我对可执行文件的生成有了更加深刻的理解，对于词法分析，语法分析，中间代码生成，目标代码生成这些流程的代码实现也有了比较详细的了解。明白了完成代码只是程序执行的第一步骤，让代码成功地运行还需要编译程序，硬件平台的配合，因此作为一个计算机专业的学生，我们更应该学习这方面的知识，以此提高自己的代码的效率。

对于本实验，我觉得难度可以稍微提高一点，可以设置一些提高实验，让有能力的同学能够从头到尾，完成一个编译程序的设计。