编译原理实验报告

专　　业：	计算机科学与技术
班　　级：	**CS2011**
学　　号：	**U202011675**
姓　　名：	徐锦慧
电　　话：	**18071680638**
邮　　箱：	**2939361916@qq.com**

原创性声明

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：徐锦慧

日期：2023 年 06 月 14 日

综合成绩	
教师签名	

目　录

1 编译工具链等

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

1.2.1 编译工具链的使用：gcc/clang/arm-linux-gnueabihf-gcc

1. GCC 编译器的使用

要求：题目已给 def-test.c、alibaba.c、alibaba.h 文件，需要在编辑器填写编译指令，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。

实现：使用 -o 参数指定输出文件，-DBLIBILI 参数进行宏定义，完成对应输出。具体代码为：

```
gcc -o def-test -DBILIBILI def-test.c alibaba.c
```

2. CLANG 编译器的使用

要求：在 shell2.sh 文件中填写指令，用 clang 编译器把所给程序 bar.c“翻译”成优化的 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。

实现：使用 -S 参数完成编译并生成汇编文件，-O2 参数控制翻译的优化级别，-target armv7-linux-gnueabihf 参数完成架构指定，-o 参数指定输出文件。具体代码如下：

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

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

要求：在 shell3.sh 文件中填写指令，用 arm-linux-gnueabihf-gcc 编译 iplusf.c

并连接 SysY2022 的运行时库 sylib.a，生成 arm 的可执行代码，最后用 qemu-arm 运行 iplusf.arm。

实现：首先用 arm-linux-gnueabihf-gcc 交叉编译器，通过 -S 参数将 iplusf.c 编译成 arm 汇编代码 iplusf.arm.s，并设置 arm 型号为 arm7；接着通过 -o 参数将 iplusf.arm.s 链接 sylib.a 生成 arm 的可执行代码 iplusf.arm；最后通过 qemu-arm -L 命令将上述生成的 iplusf.arm 可执行代码在 qemu-arm 虚拟机上执行，设置库路径为 /usr/arm-linux-gnueabihf/。具体代码为：

```
arm-linux-gnueabihf-gcc -S iplusf.c iplusf.arm.s
arm-linux-gnueabihf-gcc -o iplusf.arm iplusf.arm.s sylib.a
qemu-arm -L /usr/arm-linux-gnueabihf/ iplusf.arm
```

4. make 的使用

要求：在 Makefile 文件中填写代码，为 helloworld 目标编写一条生成一个名为 helloworld 的可执行文件的规则。

实现：首先根据 make 指令构建的对象，写出目标 halloworld；由源代码可知，该目标的依赖文件为 main.cc 和 helloworld.cc；接着写出构建目标的具体 g++ 指令，通过 -I ./include 编译选项添加头文件 helloworld.hh 的路径，通过 -o 参数生成目标文件。具体代码如下：

```
helloworld:main.cc helloworld.cc
	g++ -I ./include main.cc helloworld.cc -o helloworld
```

在执行过程中，出现如图 1.1 所示的报错。经过检查发现，是由于 educoder 平台上的 tab 被替换成了空格，导致格式不符合条件。在本地 txt 文件中完成 Makefile 并粘贴后，成功通过测评。

图 1.1 make 的使用报错

1.2.2 Sysy 语言与运行时库

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。可以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。要求设计一个算法来计算能获取的最大利润，并返回最大利润值。如果不能获取任何利润，返回 0。

实现：首先完成 maxProfit(int prices[]) 函数。best 表示最大利润值，初始化为 0，i 表示买入股票的日期，j 表示卖出日期，通过 2 个 while 循环找出最大利润值并返回。具体代码如下：

```
int maxProfit(int prices[]){
    int best=0,i=0;
    while(i<N){
        int j=i+1;
        while(j<N){
            if(best<prices[j]-prices[i]) best=prices[j]-prices[i];
            j=j+1;
        }
        i=i+1;
    }
    return best;
}
```

接着在 main() 中通过 getint() 函数输入股价数组，并完成对 maxProfit(prices) 的调用，再通过 putint() 输出结果。注意最后要用 putch() 函数输出换行符。具体代码如下：

```
int main(){
int i=0;
    while(i<N){
        prices[i] = getint(); i=i+1;
    }
    int best = maxProfit(prices);
    putint(best); putch(10);
   return 1;
}
```

1.2.3 目标平台-ARM

任务：补充 bubblesort.s 文件中的 arm 汇编代码，以完成程序中的 bubblesort 函数。

实现：

1) 将用到的寄存器 r0~r10 入栈。
2) 初始化寄存器 r2，用其控制外层循环次数。在 main()调用 bubblesort()之前，

将数组 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，并复制到寄存器 r8。

3) l1 表示外层循环开始。首先要判断外层循环次数，如果 r2>r8，就直接跳转至 stop，表示循环结束；否则外层循环次数加一，并初始化内层循环次数。

4) 进入内层循环 l2。r3 表示内层循环次数，r7 表示 r8-r2，如果 r3>r7，则跳转至外循环 l1，内层循环结束；否则用 r5 表示 a[j]的值，r6 表示 a[j+1]的值。如果 r5<r6，就执行下一层内循环；否则交换 r5 和 r6 的内容，再进行下一层内循环。

5) 最后需要用 pop 指令将寄存器 r0~r10 出栈。bubblesort 返回值为 0，由 r0 传递。

代码的流程图如图 1.2 所示。

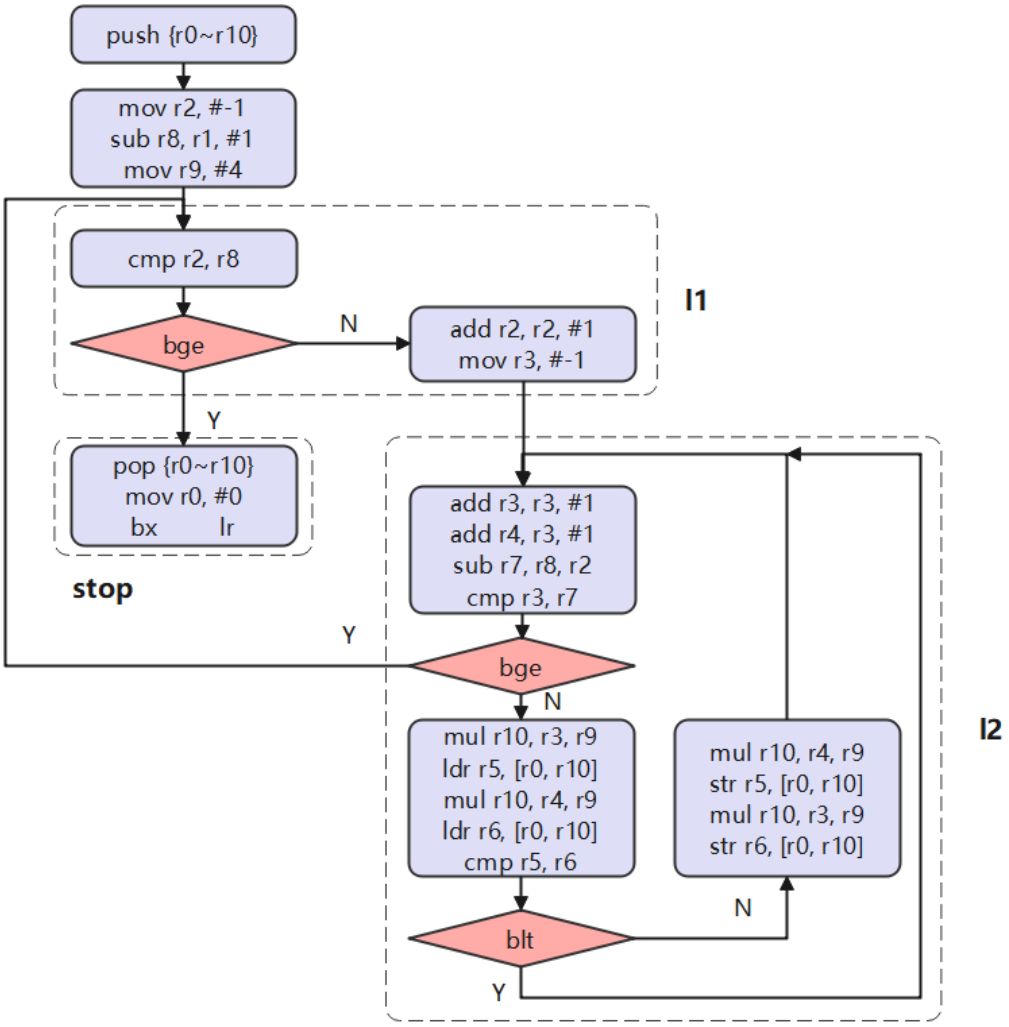

图 1.2 bubblesort 流程图

执行过程中出现了如图 1.3 所示的报错。经检查发现，是由于没有及时保存现场。加上为寄存器入栈出栈的代码后，成功通过测评。

图 1.3 bubblesort 报错

1.2.4 目标平台-RISCV

任务：用 RISCV 汇编编写一个对数组排序的函数。

实现：该关卡已完成。实现过程与上一关类似，此处略过。

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

2.2.1 基于 flex 的词法分析器(C 语言实现)

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。

实现：flex 是通过处理其源文件来生词法和语法分析器的，源文件的扩展名为.l，其语法被分为如下三个部分，将分别介绍这 3 个部分的实现。

```
/* 辅助定义部分 */
%option yylineno noyywrap
%{C 代码}%
正则式定义
%%
/* 规则部分 */
正则式  {对应的动作：C 代码}
%%
/* 用户子程序段 */
```

1) 辅助定义部分

这一部分一般是一些声明及选项设置等。C 语言的注释、头文件包含等一般就放在 %{ %} 之间，这一部分的内容会被直接复制到输出文件的开头部分，定义段还可以定义正则式，这些正则式可以在其定义后被后续的正则式所引用。引用之前定义的正则式，要用{}将先前定的的正则式括起来。

定义标识符 ID。根据 SysY2022 语言定义，标识符的由字母、数字、'_'组成，且首字符必须为字母或'_'。具体实现为：

```
ID [a-zA-Z_][a-zA-Z0-9_]*
```

定义 int 型字面量 INT_LIT。整型常量的定义如图 2.1 所示，可以为八进制、

十进制、十六进制。其中，十进制数不能以 0 开头（数字 0 除外）；八进制数以 0 开头且每位数字不能超过 7；十六进制数以 0x 或 0X 开头，且每位数字由 a-fA-F 及 0-9 范围内的字符组成。具体实现为：

```
INT_LIT [1-9][0-9]*|0[0-7]*|(0x|0X)[0-9a-fA-F]+
```

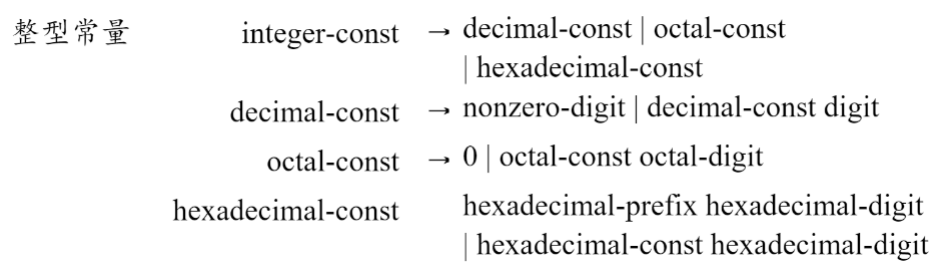

图 2.1 整型常量

定义 float 型字面量 FLOAT_LIT。浮点型数由 4 部分组成：①小数点前的数字，定义为[0-9]*；②小数点符号，定义为[.]；③小数点后的数字，定义为[0-9]*；④小数点后的数字可以再加上科学计数，例如 4e-04，定义为([Ee][+-]?[0-9]+)?|[0-9]+([Ee][+-]?[0-9]+)；⑤浮点数最后也可以选择带上 f 或 F，定义为(f|F)。具体实现如下：

```
FLOAT_LIT ([0-9]*[.][0-9]*([Ee][+-]?[0-9]+)?|[0-9]+([Ee][+-]?[0-9]+))(f|F)?
```

定义注释。单行注释在识别到“//”后开始，在识别到换行符结束；多行注释则从识别到“/*”开始，一直到“/*”结束。

```
MultilineComment "/*"(.|\n)*"*/"
SingleLineComment "//".*
```

定义词法错误。当以数字开头，且后续有字母或’_’时，将字符串识别为非法标识符或非法 float 型数字；当数字以 0 开头且后面的数字超过 7 时，则是错误的八进制表示，识别为非法 int 型数字。具体实现如下：

```
Invalid [0-9]+[a-zA-z_]+[a-zA-Z_0-9]*
Invalid_int 0[0-9]+
```

2) 规则部分

该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：模式 1 {动作 1 (C 代码)}，在输入和模式 1 匹配的时候，执行动作部分的代码；即便对应的动作为空，也应该使用{}。

对于该部分实现，可分成单关键字和用户自定义单词的识别。对单关键字直接进行识别，例如识别 int 关键字， 具体实现为：

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

而对于用户自定义单词，如标识符 ID、int 型字面量 INT_LIT、float 型字面量 FLOAT_LIT，可以引用辅助部分的相关定义，使用引用方式进行识别，具体实现为：

```
{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;}
```

对于错误单词识别，则可使用上述定义的非法标识符识别与数值识别。具体实现为：

```
{Invalid} {printf("Lexical error - line %d : %s\n",yylineno,yytext); return LEX_ERR;}
{Invalid_int} {printf("Lexical error - line %d : %s\n",yylineno,yytext); return LEX_ERR;}
```

3) 用户子程序段

本部分为 C 代码，会被原样复制到输出文件中，一般这里定义一些辅助函数等，如动作代码中使用到的辅助函数。此处题目已给出，不用再添加代码。

2.2.2 基于 flex 的词法分析器(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.cpp，词法分析器能输出该程序的 token 以及 token 的种别。

实现：该关卡已完成。实现过程与上一关类似，此处略过。

2.2.3 基于 antlr 的词法分析器(C++实现)

任务：利用 antlr 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.cpp，词法分析器能输出该程序的 token 以及 token 的种别。

实现：

1) 在 SysyLex.g4 文件中，补充缺少的词法规则。

标识符 ID 的定义。ID 由数字、字母、’_’组成，且首字符是字母或’_’。具体实现如下：

```
// identifier
ID : [a-z_A-Z][a-z_A-Z0-9]*
   ;
```

int 型字面量 INT_LIT 的定义。int 型字面量分为八进制数、十进制数、十六进制数，其中十进制不能以 0 开头；八进制以 0 开头，且每位数字不能超过 7；

十六进制以 0x 或 0X 开头，每位数字由 a-f 或 A-F 或 0-9 组成。具体实现如下：

```
// integer literal
INT_LIT : [1-9][0-9]*
        | '0'[0-7]*
        | '0x'[0-9a-fA-F]+
        | '0X'[0-9a-fA-F]+
        ;
```

float 型字面量 FLOAT_LIT 的定义。float 型字面量由带小数点的数组成，后面可以带有 eE 表示的科学计数和 fF 后缀。具体实现如下：

```
// float literal
FLOAT_LIT : [+-]?(('.'[0-9]+([eE][+-]?[0-9]+)?[fF]?)
        |([0-9]+'.'[0-9]*([eE][+-]?[0-9]+)?[fF]?)
        |([0-9]+[eE][+-]?[0-9]+[fF]?))
        ;
```

词法错误的定义。由提示可知，数位上有超过 7 的八进制数、以数字开头的标识符都不是合法的字面量。具体实现如下：

```
// error
LEX_ERR : '0'[0-9]*[8-9]+ [0-9]*
        | [0-9a-zA-Z]*
        ;
```

2) 在 main.cpp 中，正确显示 token 及 token 的种别信息。

首先，通过 getType()得到识别出的单词种类 tokentype。如果是合法的单词，则直接输出“识别出的单词：种别名称”；如果是词法错误，则报告其错误类别和位置，报告格式为“Lexical error - 行号：识别出来的串”；对于 whitespace 和注释，则直接忽略。具体实现如下：

```
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;
            }
```

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

3.2.1 基于 flex/bison 的词法分析器(C 语言实现)

1. Sysy2022 语法检查

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，能识别并定位源程序中的语法错误。

实现：

1) 转换成 bison 语法规则，符号'='、';'、'if'等用相应的 Token 代号取代。例如'='替换成 ASSIGN，';'替换成 SEMICOLON。
2) 类似如下含有任选项 Exp 的产生式：Stmt → [Exp] ';'，应将其改为两个产生式：Stmt → ';' //Exp 不出现；Stmt → Exp ';' //Exp 可以出现一次。
3) 在每个语句后加上语义动作。由于本关只需要语法检查，因此语义动作可以都设置为{ $$ = NULL; }。具体代码如下：

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

2. SysY2022 语法分析

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)。

实现：语义规则部分在第一关中已实现，此处只需要为每条语句补充对应的语义动作，即调用 new_node()函数创建 AST。

以 LVal ASSIGN Exp SEMICOLON 为例。该语句的类型为 AssignStmt，子节点分别为$1,$3，放在 left 和 right 的位置；如果只有一个子节点，则放在 right 的位置；如果有三个子节点，放在 left,mid,right 的位置。据此可以完成所有语句的语义动作，具体实现如下：

```
Stmt: LVal ASSIGN Exp SEMICOLON {
            $$ = new_node(Stmt, NULL, $1, $3, AssignStmt, 0, NULL, NonType);}
        | SEMICOLON {
            $$ = new_node(Stmt, NULL, NULL, NULL, BlankStmt, 0, NULL, NonType);}
        | Exp SEMICOLON {
            $$ = new_node(Stmt, NULL, NULL, $1, ExpStmt, 0, NULL, NonType);}
        | Block {
            $$ = new_node(Stmt, NULL, NULL, $1, Block, 0, NULL, NonType);}
        | IF LP Cond RP Stmt {
            $$ = new_node(Stmt, $3, NULL, $5, IfStmt, 0, NULL, NonType);}
        | IF LP Cond RP Stmt ELSE Stmt {
            $$ = new_node(Stmt, $3, $5, $7, IfElseStmt, 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);};
```

3.2.2 基于 flex/bison 的词法分析器(C++实现)

1. SysY2022 语法检查（C++)

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，能识别并定位源程序中的语法错误。

实现： 本关实现过程和 3.2.1 中的第一关类似，只需把$$ = NULL 换成$$ = nullptr。代码如下：

```
Stmt: LVal ASSIGN Exp SEMICOLON { $$ =  nullptr; }
        | SEMICOLON { $$ = nullptr; }
        | Exp SEMICOLON { $$ = nullptr ; }
        | Block { $$ = nullptr; }
```

```
    | IF LP Cond RP Stmt { $$ = nullptr; }
    | IF LP Cond RP Stmt ELSE Stmt { $$ = nullptr; }
    | WHILE LP Cond RP Stmt { $$ = nullptr; }
    | BREAK SEMICOLON { $$ = nullptr; }
    | CONTINUE SEMICOLON { $$ = nullptr; }
    | RETURN Exp SEMICOLON { $$ = nullptr; }
    | RETURN SEMICOLON { $$ = nullptr; };
```

2. SysY2022 语法分析

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)。

实现： 语法规则部分已由第一关实现，此处只需补充每条语句的语义动作。以 LVal ASSIGN Exp SEMICOLON 为例，首先用 StmtAST()构造语法树，由 ASSIGN 可知 sType 的值为 ASS；lVal 在表达式中的位置为 1，其值为 unique_ptr<LValAST>($1)；exp 的位置为 3，值为 unique_ptr<AddExpAST>($3)。需要注意此处没有 ExpAST，需要用 AddExpAST。以此类推，可得全部语句的代码实现：

```
Stmt : LVal ASSIGN Exp SEMICOLON {
            $$ = new StmtAST();
            $$->sType = ASS;
            $$->lVal = unique_ptr<LValAST>($1);
            $$->exp = unique_ptr<AddExpAST>($3);}
     | SEMICOLON {
            $$ = new StmtAST();
            $$->sType = SEMI;}
     | Exp SEMICOLON {
            $$ = new StmtAST();
            $$->sType = EXP;
            $$->exp = unique_ptr<AddExpAST>($1);}
     | Block {
            $$ = new StmtAST();
            $$->sType = BLK;
            $$->block = unique_ptr<BlockAST>($1);}
     | SelectStmt {
            $$ = new StmtAST();
            $$->sType = SEL;
            $$->selectStmt = unique_ptr<SelectStmtAST>($1);}
     | IterationStmt{
            $$ = new StmtAST();
            $$->sType = ITER;
            $$->iterationStmt = unique_ptr<IterationStmtAST>($1);}
     | BREAK SEMICOLON {
            $$ = new StmtAST();
            $$->sType = BRE;}
```

```
    | CONTINUE SEMICOLON {
        $$ = new StmtAST();
        $$->sType = CONT;}
    | ReturnStmt{
        $$ = new StmtAST();
        $$->sType = RET;
        $$->returnStmt = unique_ptr<ReturnStmtAST>($1);}
    ;
```

注意 IF 语句中不能直接使用 SelectStmtAST，会导致类型不匹配的错误。因此，需要在后面自己定义一个 SelectStmt 类型，代码如下：

```
SelectStmt : IF LP Cond RP Stmt ELSE Stmt {
        $$ = new SelectStmtAST();
        $$->cond = unique_ptr<LOrExpAST>($3);
        $$->ifStmt = unique_ptr<StmtAST>($5);
        $$->elseStmt = unique_ptr<StmtAST>($7);}
    |   IF LP Cond RP Stmt {
        $$ = new SelectStmtAST();
        $$->cond = unique_ptr<LOrExpAST>($3);
        $$->ifStmt = unique_ptr<StmtAST>($5);
        $$->elseStmt = nullptr;}
    ;
```

WHILE 语句的实现需要自己定义 IterationStmt 来实现，代码如下：

```
IterationStmt : WHILE LP Cond RP Stmt {
        $$ = new IterationStmtAST();
        $$->cond = unique_ptr<LOrExpAST>($3);
        $$->stmt = unique_ptr<StmtAST>($5);}
    ;
```

RETURN 语句的实现需要自己定义 ReturnStmt 来实现，代码如下：

```
ReturnStmt : RETURN SEMICOLON {
        $$ = new ReturnStmtAST();
        $$->exp = nullptr;}
    |   RETURN Exp SEMICOLON {
        $$ = new ReturnStmtAST();
        $$->exp = unique_ptr<AddExpAST>($2);}
    ;
```

3.2.3 基于 antlr 的词法分析器(C++实现)

1. 用 antlr 生成 Sysy2022 语言的语法分析器

任务：利用 antlr 生成 Sysy2022 的语法分析程序。要求任给一个 Sysy2022 语言的有语法错误源程序，能够指出错误在哪一行出现,并提示错误信息。

实现：

1) 'while' '(' Cond ')' Stmt 的转换。首先根据 SysyLex.g4 中的语法规则，将'while'转换成 While，'('转换成 Lparen，Cond 转换成 cond，')'转换成 Rparen，Stmt 转换成 stmt，产生式后用# while 标注标签，表明这是一个 while 语句。
2) 'break' ';的转换。根据语法规则，'break'转换成 Break，';'转换成 Semicolon，产生式后用# break 标注，表明这是一个 break 语句。
3) 'continue' ';'的转换。根据语法规则，'continue'转换成 Continue，';'转换成 Semicolon，产生式后用# continue 标注，表明这是一个 continue 语句。
4) 'return' [Exp] ';'的转换。根据语法规则，'return'转换成 Return，[Exp]表示可由可无，转换成(exp)?，';'转换成 Semicolon，产生式后用# return 标注，表明这是一个 return 语句。具体代码如下：

```
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
    ;
```

2. 用 antlr 生成 Sysy2022 语言的 AST

任务：利用 antlr 生成 Sysy2022 语言的语法分析程序，要求任给一个语法正确的 Sysy2022 语言的源程序，输出其抽象语法树(AST)。

实现：首先观察 WhileContex 在 SysyParser.h 中的定义，添加 cond 和 stmt 参数，并对其进行强制类型转换。再通过 new While()新建对象并返回。具体实现如下：

```
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);
```

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) 在访问 lVal 之前通过全局临时变量 requireLVal 传递信息，表示当前的 lVal 是赋值语句左值，不是表达式。
2) 通过 accept()访问 IVal。
3) 在访问 IVal 之后从 recentVal 取左值的 Value。
4) 通过 accept()访问 exp。
5) 在访问 exp 后，从 recentVal 取右值的 Value。
6) 检查赋值语句左值和右值的类型。由于 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，当两个操作数类型不同时要做强制类型转换。
7) 调用 IRStmtBuilder::create_store()方法生成 store 指令。具体实现如下：

```
    requireLVal = true;            // 表示当前的 lVal 是赋值语句左值，不是表达式
    ast.lVal->accept(*this);       // 访问 IVal
    auto val1 = recentVal;         // 从 recentVal 取左值
    is_single_exp = true;
    ast.exp->accept(*this);        // 访问 exp
    auto val2 = recentVal;         // 从 recentVal 取右值
    // 检查赋值语句左值和右值的类型，必要时作类型转换
    if (val1->type_->tid_ == Type::FloatTyID && val2->type_->tid_ == Type::IntegerTyID) {
        val2 = builder->create_sitofp(val2,FLOAT_T);
    }
    else if (val1->type_->tid_ == Type::IntegerTyID && val2->type_->tid_ ==
Type::FloatTyID){
        val2 = builder->create_fptosi(val2, INT32_T);
    }
    builder->create_store(val2, val1);
```

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

5.2.1 基于 LLVM 的目标代码生成(ARM)

任务：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。允许调用 LLVM 的库函数实现上述功能。

实现：

1) 初始化目标的 registry

在 codegen.cc 文件中初始化所有用来生成目标代码的目标，具体实现如下：

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

auto target_triple = "armv7-unknown-linux-gnueagihf";
```

2) 设置目标平台

此处还需要一个目标机器 (TargetMachine)，这个类提供了我们将要生成的代码的目标平台的完整机器描述。在本任务中，为了简化问题，我们只需要使用通用 CPU，不需要任何附加功能、选项或重定位模型。具体实现如下：

```
auto cpu = "generic";
auto features = "";
TargetOptions opt;
auto RM = Optional<Reloc::Model>();
auto TheTargetMachine =
        target->createTargetMachine(target_triple, cpu, features, opt, RM);
module->setDataLayout(TheTargetMachine->createDataLayout());
```

3) 初始化 addPassesToEmitFile()函数的参数

① 首先调用 getGenFilename()函数，获得要写入的目标代码文件名 filename。

② 接着声明 std::error_code 类型的对象 EC，并通过构造函数实例化 raw_fd_ostream 类的对象 dest，Flags 置 sys::fs::OF_None。在调用函数后检查 EC，EC 为 true 则表明有错误发生，此时应该输出提示信息后返回。

③ 实例化 legacy::PassManager 类的对象 pass。

④ 为 file_type 赋初值。具体代码如下：

```
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 = CGFT_AssemblyFile;
```

5.2.2 基于 LLVM 的词法分析器(RSICV64)

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)。

实现：

1) 初始化目标的 registry

在 codegen.cc 文件中初始化所有用来生成目标代码的目标，设置 target_triple 为"riscv64-unknown-elf"。具体实现如下：

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

auto target_triple = "riscv64-unknown-elf";
```

2) 指定目标平台

此处还需要一个目标机器 (TargetMachine)，这个类提供了我们将要生成的代码的目标平台的完整机器描述。在本任务中，为了简化问题，只需要使用通用 CPU，不需要任何附加功能、选项或重定位模型。具体实现如下：

```
auto cpu = "generic-rv64";
auto features = "";

TargetOptions opt;
auto RM = Optional<Reloc::Model>();
auto TheTargetMachine =
```

```
target->createTargetMachine(target_triple, cpu, features, opt, RM);
module->setDataLayout(TheTargetMachine->createDataLayout());
```

3) 初始化 addPassesToEmitFile()函数的参数

此处实现与 5.2.1 类似，具体代码如下：

```
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;
```

6 总结

6.1 实验感想

本次实验难度较小，由于老师给出了详细的注释和指导，总体来说基本可以完成。

通过参与编译原理课程的实验，我深刻体会到编译器在软件开发中的重要性。编译原理不仅是一门理论课程，更是一门需要实践的学科。通过亲自编写和运行代码，我对编译过程的五个阶段有了更深入的理解。

在实验过程中，我也遇到了一些挑战和困难，例如设计合适的语法规则和语法动作、构建抽象语法树等。通过仔细学习文档、查阅资料、与同学们讨论和实践调试，我们逐渐克服了这些困难，并成功完成了实验任务。

此外，实验也让我明白了编译原理在软件开发中的价值。编译器是将高级语言转换为机器码的关键工具，它能够提高代码的运行效率和可移植性。了解和掌握编译原理技术，对我们未来的职业发展将起到重要的推动作用。

非常感谢课题组老师们的辛苦付出，亲自设计了实验流程、指导书以及讲解视频。对此也有一些小建议：一是希望可以稍微提高难度，减少一些提示。由于很多关卡注释和视频讲解都直接给出了代码，会有种什么都不会就做完实验的感觉，所以感觉实验难度上可以再优化些。二是希望实验指导书可以更清晰易懂些，尤其是在做四、五关时，有种云里雾里的感觉，头歌左侧的说明也可以加入指导书中，这样更方便查看。

6.2 实验总结与展望

本次编译原理课程的实验主要分为编译工具的使用、词法分析过程、语法分析过程、中间代码生成、目标代码生成五个部分，通过学习和实践，我对他们有了更深入的了解。

通过使用编译工具链，我了解了编译器的各个阶段和相互之间的工作关系。这些工具不仅帮助我们分析代码、检测错误，还能生成可执行的目标代码，对于编译器的开发和调试非常有帮助。

通过学习 Sysy 语言及其运行时库，我掌握了简单的静态类型语言的基本语法、类型系统和流程控制结构，了解了编译器如何将高级语言转换为中间代码，并在运行时环境中执行。

通过学习目标平台 ARM 和 RISC-V64 的汇编语言，我熟悉了它们的指令集、寄存器分配和内存管理等特性。这让我对底层硬件和汇编语言有了更深入的了解，并为后续的目标代码生成和优化打下了基础。

通过实现 Sysy 语言的语法分析器和词法分析器，我深入了解了 Sysy 语言的语法规则和词法分析器的工作原理，掌握了不同工具和语言的使用方法，并通过实践加深了对编译器工作流程中语法分析阶段的理解。

通过将 Sysy 语言程序翻译成 LLVM IR 中间代码，我深入了解了 LLVM IR 的语法和中间代码生成的原理，并掌握了如何设计合适的代码生成策略、处理语法和语义错误等。

总之，编译原理课程的实验让我受益匪浅。通过实践，我不仅学到了编译器的工作原理和技术，还培养了解决问题和团队合作的能力。我相信这些知识和经验对我未来的学习和职业发展将会产生积极的影响。

参考文献

[1] 袁春风，余子濠. 计算机系统基础（第 2 版）. 北京：机械工业出版社，2018

[2] 王生原，董渊. 编译原理（第 3 版）. 北京：清华出版社， 2015

[3] SysY2022 语言定义.pdf

[4] https://efxa.org/2014/05/25/how-to-create-an-abstract-syntax-tree-while-parsing-an-input-stream/